TCP/IP协议及网络编程技术

罗军舟　黎波涛　杨明　吴俊　黄健　编著



清华大学出版社

TCP/IP 协议及网络编程技术

罗军舟　黎波涛　杨明　吴俊　黄健　编著

清华大学出版社

北　京

内 容 简 介

本书是一本关于TCP/IP协议原理及编程技术的教材，由两部分组成：第1部分系统地介绍了TCP/IP协议族的体系结构，并分别介绍了IP层、传输层和应用层中各种协议的具体原理和工作机制；第2部分介绍了基于socket编程接口的网络编程技术，重点讲述了客户端和服务器端编程应注意的问题、可用的模式和技术。

作为一本TCP/IP协议理论基础和编程技术的教材，本书既注重清晰地描述概念和理论，又做到了理论联系实际，能有效提高读者对TCP/IP的理解和网络开发能力。它可作为高等院校计算机、通信等专业的教学参考书，也可供从事相关开发工作和网络管理的人员参考。

图书在版编目（CIP）数据

TCP/IP协议及网络编程技术/罗军舟，黎波涛，杨明编著. —北京：清华大学出版社，2004.10
ISBN 7-302-09558-2

Ⅰ. T…　Ⅱ. ①罗…　②黎…　③杨…　Ⅲ. ①计算机网络-通信协议-教材　②计算机网络-程序设计-教材　Ⅳ. ①TN915.04　②TP393.09

中国版本图书馆CIP数据核字（2004）第095660号

出 版 者：清华大学出版社　　**地　　址：**北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编：**100084
社 总 机：010-62770175　　**客户服务：**010-62776969
责任编辑：马　丽
封面设计：秦　铭
版式设计：俞小红
印 刷 者：
装 订 者：
发 行 者：新华书店总店北京发行所
开　　本：185×260　**印张：**24　**字数：**530千字
版　　次：2004年10月第1版　2004年10月第1次印刷
书　　号：ISBN 7-302-09558-2/TP・6648
印　　数：1~5000
定　　价：29.00元

前　　言

Internet 是 20 世纪最伟大的发明之一，它将全世界数以千万计的计算设备（不管它们是庞大的巨型机，还是桌面上的个人电脑，甚至是人们口袋中的移动电话）连接成一个巨大的网络，并使它们能够在彼此之间迅速方便地传输信息。整个世界好像突然变小了，不同地区的人与人之间的距离不再遥不可及。然而，改变整个世界的不只是 Internet 本身，还有无法计数的构筑在其上的应用软件。通过电子邮件，信件的往来不再需要几天甚至几周了；通过网上商城，在家就可以购物；视频点播让人们可以在家中欣赏喜爱的电影。如果没有这些应用，Internet 至今还仅仅是科研人员实验室里使用的科研工具。

正如 Internet 的核心 TCP/IP 协议的目标所指出的，任何人都可以方便地使用 Internet，并在其上开发出新的应用。当然，要开发基于 Internet 上的应用必须先知道它是如何工作的，即它是如何将各种不同的设备连接起来的，如何将数据从一个计算设备传输到另一个的，是如何支撑各种各样的应用软件的。当然，如果你的工作不需要知道这些，如果你对此不感兴趣，那就可以合上这本书了。但如果你是一个程序员，或者你想成为他们中的一员；如果你正在开发一个网络软件，你开发的软件的客户端或服务器端在局域网工作正常但在广域网中却总出问题，或者它们的效率总达不到要求；如果你是所在单位的网络管理员，那么本书将会对你有所帮助。

司机虽然不用生产自己开的汽车，但一个好的司机应该知道汽车的工作原理。同样，网络软件开发人员不用自己设计通信网络的协议，但应该知道网络协议的工作原理和机制，这样才能开发出正确、稳定、高效的网络软件。本书的目的是帮助读者提高对 Internet 的理解和网络编程能力。为达到这个目的，本书从 Internet 的工作原理 TCP/IP 协议族和实际的编程模式和技巧两个方面进行了介绍。

本书由两部分组成：

第 1 部分由 1~16 章组成，介绍了 TCP/IP 协议族的体系结构及各层组成协议的工作机制。这部分介绍的各种协议是网络编程中常见的需要了解的协议，对它们的理解有助于理解各种网络编程技术。第 1 章介绍了 Internet 的发展历史、现状及发展趋势；第 2 章对 TCP/IP 协议族总体的体系结构进行了系统的说明；第 3~7 章介绍网络层中的部分重要协议，其中重点是作为 TCP/IP 核心的 IP 协议；第 8 章和第 9 章分别介绍了传输层的两个协议：UDP 和 TCP；第 10~14 章介绍几种常见的应用层协议，包括远程登录、电子邮件、HTTP 协议、网络文件和网络管理等；第 15 章对下一代 IP 协议即 IPv6 进行了介绍；第 16 章则简单介绍了常见操作系统（Windows，UNIX/Linux）中的 TCP/IP 协议的实现机制。

第 2 部分由 17~22 章组成，介绍了网络编程的接口、模式和技巧。第 17 章和 19 章详细介绍了 Windows 中的网络编程接口 Socket 及使用；第 18 章、第 20 章和 21 章介绍客户端/服务器的网络编程模型，并重点介绍了服务器端编程经常使用的技术，第 22 章通过对

一个完整的 Web 服务器程序的分析，进一步说明了这部分各章中介绍的各种编程技术。

通过对各种协议工作机制的了解，并使用试验验证各种网络编程技术，理论和实践紧密结合，相信读者对 Internet 的理解和编程能力都能在较短时间内得到提高。如果本书确实能够为读者提供帮助，那将是我们最大的荣幸。

由于时间仓促，加之作者水平有限，书中难免会有不足之处，真诚欢迎各位读者予以批评指正。

作　者

2004 年 6 月

目　　录

第 1 章　Internet 概述

什么是 Internet？在英语中“Inter”的含义是“交互的”，“net”是指“网络”。简单而言，Internet 是指一个由计算机构成的交互网络。它是一个世界范围内的巨大的计算机网络体系，它把全球数万个计算机网络，数千万台主机连接起来，包含了难以计数的信息资源，向全世界提供信息服务。它的出现，是世界由工业化走向信息化的必然和象征，但这并不是对 Internet 的一种定义，而仅仅是对它的一种解释。从网络通信的角度来看，Internet 是一个以 TCP/IP 网络协议连接各个国家、各个地区、各个机构的计算机网络的数据通信网。从信息资源的角度来看，Internet 是一个集各个部门、各个领域的各种信息资源为一体，供网上用户共享的信息资源网。现在的 Internet 已经远远超过了一个网络的涵义，它是一个信息社会的缩影。虽然至今还没有一个准确的定义来概括 Internet，但是这个定义应从通信协议、物理连接、资源共享、相互联系、相互通信等角度来综合加以考虑。

了解一个事物的最有效方法莫过于先了解它的历史，在本章中，先简要回顾一下 Internet 的发展历史，再介绍与 Internet 相关的管理结构，并对当前的 Internet 应用现状与发展趋势作一简单介绍。

1.1　Internet 发展历史

Internet 最早来源于美国国防部高级研究计划局 DARPA（Defense advanced Research Projects Agency）的前身 ARPA 建立的 ARPAnet，该网于 1969 年投入使用。从 20 世纪 60 年代开始，ARPA 就开始向美国国内大学的计算机系和一些公司提供经费，以促进基于分组交换技术的计算机网络的研究。1968 年，ARPA 为 ARPAnet 网络项目立项，该项目基于这样一种主导思想：网络必须能够经受住故障的考验而维持正常工作，一旦发生战争，当网络的某一部分因遭受攻击而失去工作能力时，网络的其他部分应当能够维持正常通信。最初，ARPAnet 主要用于军事研究目的，它有五大特点：

（1）支持资源共享。

（2）采用分布式控制技术。

（3）采用分组交换技术。

（4）使用通信控制处理机。

（5）采用分层的网络通信协议。

1969 年 6 月，完成第一阶段的工作，组成了 4 个结点的试验性网络，称为 ARPAnet。ARPAnet 采用称之为接口报文处理器（IMP）的小型机作为网络的结点机，为了保证网络的可靠性，每个 IMP 至少和其他两个 IMP 通过专线连接，主机则通过 IMP 接入 ARPAnet。IMP 之间的信息传输采用分组交换技术，并向用户提供电子邮件、文件传送和远程登录等服务。

ARPAnet 被公认为世界上第一个采用分组交换技术组建的网络。

1972 年，ARPAnet 在首届计算机后台通信国际会议上首次与公众见面，并验证了分组交换技术的可行性，由此，ARPAnet 成为现代计算机网络诞生的标志。

1973 年，美国国防部高级研究计划局 DARPA 正式启动并实施了一研究项目，称为“The Internetting Project”。该项目着眼于互联各种基于分组交换技术的计算机网络，并设计出一类通信协议以便于在网络计算机中透明地交互。由该项目构建的网络可视为现在 Internet 的前身，其所研发的通信协议最终发展成为著名的 TCP/IP 协议族。

1980 年，ARPA 投资把 TCP/IP 加进 UNIX（BSD4.1 版本）的内核中，在 BSD4.2 版本以后，TCP/IP 协议即成为 UNIX 操作系统的标准通信模块，这其中美国国防部的作用功不可没。

1982 年，Internet 由 ARPAnet、MILNET 等几个计算机网络合并而成，作为 Internet 的早期骨干网，ARPAnet 试验并奠定了 Internet 存在和发展的基础，较好地解决了异种机网络互联的一系列理论和技术问题。

1983 年，ARPAnet 分裂为两部分：ARPAnet 和纯军事用的 MILNET。该年 1 月，ARPA 把 TCP/IP 协议作为 ARPAnet 的标准协议。其后，人们称呼这个以 ARPAnet 为主干网的网际互联网为 Internet，TCP/IP 协议族便在 Internet 中进行研究、试验，并改进成为使用方便、效率极好的协议族。

1986 年，美国国家科学基金会 NSF（National Science Foundation）建立了 6 大超级计算机中心，为了使全国的科学家、工程师能够共享这些超级计算机设施，NSF 建立了自己的基于 TCP/IP 协议族的计算机网络 NSFnet。NSF 在全国建立了按地区划分的计算机广域网，并将这些地区网络和超级计算中心相连，最后将各超级计算中心互联起来。地区网的构成一般是由一批在地理上局限于某一地域，在管理上隶属于某一机构或在经济上有共同利益的用户的计算机互联而成。连接各地区网上主通信结点计算机的高速数据专线构成了 NSFnet 的主干网，这样，当一个用户的计算机与某一地区相连以后，它除了可以使用任一超级计算中心的设施，可以同网上任一用户通信外，还可以获得网络提供的大量信息和数据。这一成功使得 NSFnet 于 1990 年 6 月彻底取代了 ARPAnet 而成为 Internet 的主干网。

到了 20 世纪 90 年代，美国政府意识到仅靠政府资助，难以适应应用的发展需求，故鼓励商业部门介入。MCI、IBM 和 MERIT 公司联合组建 ANS（高级网络和服务公司），建立覆盖全美的、T3（44.746M）的 ANSNET，连接 ARPANET 和 NSFNET。随后，DARPA 和 NSF 撤销对 ARPAnet、NSFNET 的资助，因特网开始商用。商业机构的介入，出现大量的 ISP 和 ICP，丰富了 Internet 的服务和内容。美国政府通过因特网发布世界各国的经济、贸易信息。

Internet 的发展时间表如图 1.1 所示，图中给出了在 Internet 发展中涉及到的重大事件。

1.2　Internet 管理机构

Internet 的发展和正常运转需要一些管理机构的管理，如 IP 地址的分配需要有 IP 地址资源的管理机构，各种标准的形成需要有专门的技术管理机构。本节将介绍 Internet 各个管理

机构的职能及它们之间的关系。

ARPANET　INTERNET 雏形　INTERNET 早期骨干网　NSF　当今的 Internet

1968年　1980年　1986年　1993年

APARNET实验网　研究出TCP/IP　APARNET过渡到TCP/IP　MILNET/APARNET分离　NSF 骨干网启动　NSF net取代APARNET　World Wide Web

接入 Internet 的网络数量　3　20　60　300　500　900　19，000

图 1.1　Internet 发展时间表

1.2.1　Internet 管理机构

Internet 工作委员会（Internet Activities Board，IAB）成立于 1980 年，属于非营利机构，负责技术的方针和策略的拟定，以及管理工作的导引协调，例如有关 TCP/IP 的发展、决定哪些协议能成为 TCP/IP 的一员、在何时可以成为标准，以及因特网的演进、网络系统与通信技术的研发等工作。在 IAB 之下，有研究小组及工作小组两个主要单位，并有一些小型指导群，共同进行设定标准及决定策略的工作。IAB 的组织架构可用图 1.2 来说明。

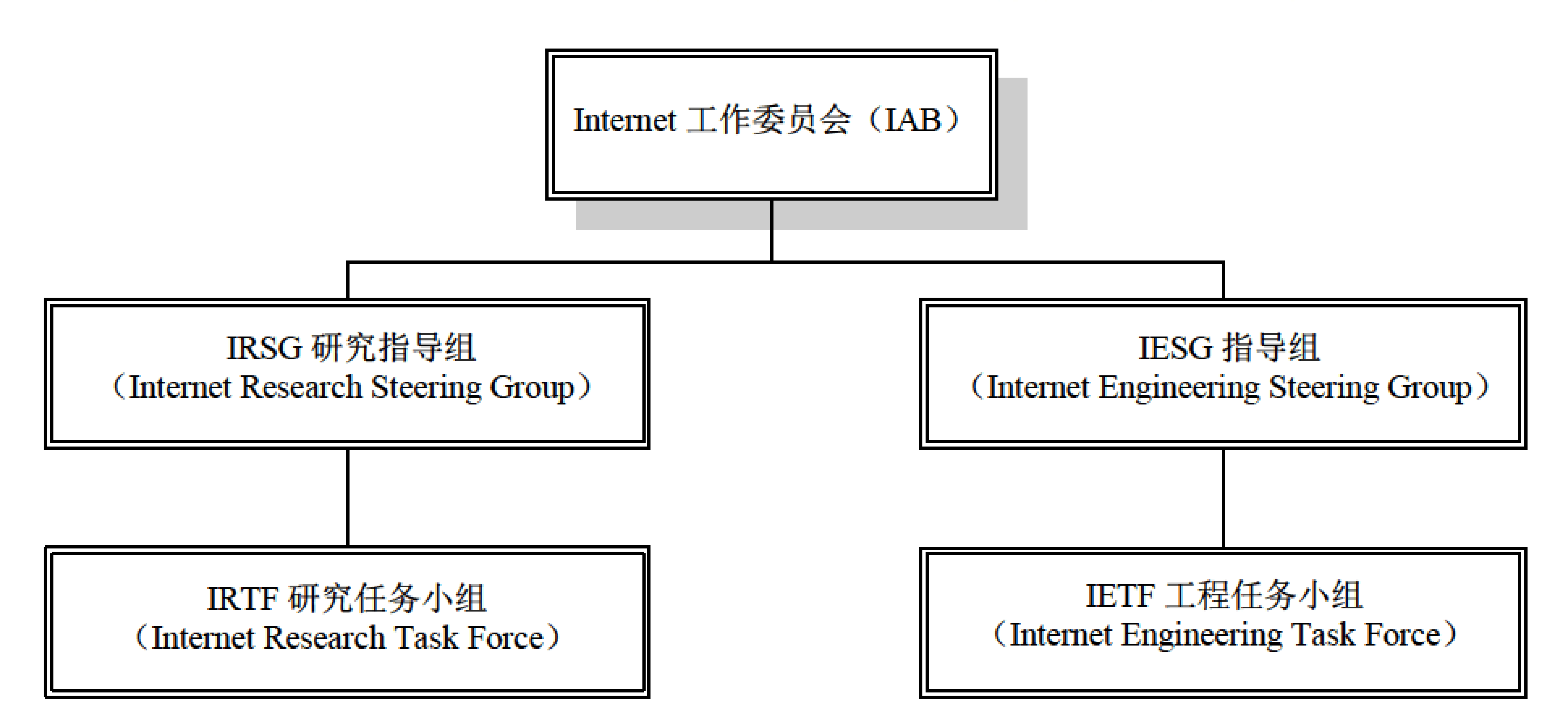


图 1.2　IAB 的组织架构图

1.2.2　Internet 域名与地址管理机构

Internet 域名与地址管理机构（ICANN）是为承担域名系统管理、IP 地址分配、协议参数配置以及主服务器系统管理等职能而设立的非营利机构。现由 IANA 和其他实体与美国政

府约定进行管理。ICANN 理事会是 ICANN 的核心权力机构，共由 19 位理事组成：9 位 At-Large 理事，9 位来自 ICANN 3 家支持组织提名的理事（每家 3 名）和一位总裁。根据 ICANN 的章程规定，它设立 3 个支持组织，从 3 个不同方面对 Internet 政策和构造进行协助、检查以及提出建议。这些支持组织帮助促进了 Internet 政策的发展，并且在 Internet 技术管理上鼓励多样化和国际参与。每家支持组织向 ICANN 董事会委派 3 位董事。这 3 个支持组织是：

（1）地址支持组织（ASO），负责 IP 地址系统的管理。

（2）域名支持组织（DNSO），负责互联网上的域名系统（DNS）的管理。

（3）协议支持组织（PSO），负责涉及 Internet 协议的惟一参数的分配。此协议是允许计算机在因特网上相互交换信息，管理通信的技术标准。

1.2.3 IP 地址管理机构

全世界国际性的 IP 地址管理机构有 4 个，即 ARIN、RIPE、APNIC 和 LACNIC，它们负责 IP 地址的地理区域，如图 1.3 所示。

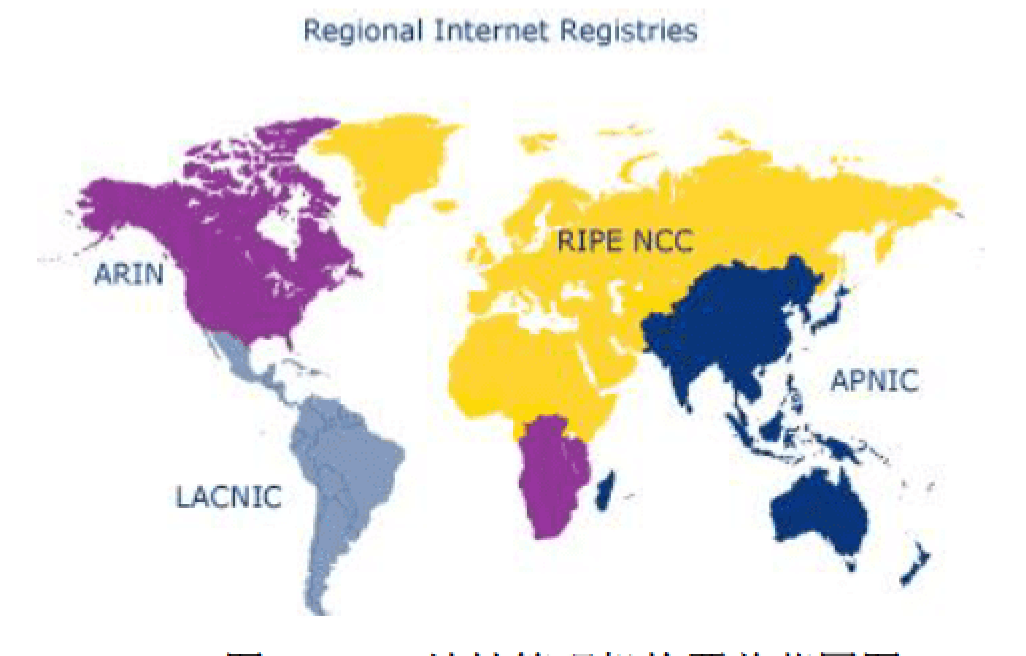


图 1.3 IP 地址管理机构覆盖范围图

其中美国 Internet 号码注册中心 ARIN（American Registry for Internet Numbers）提供的查询内容包括了全世界早期网络及现在的美国、加拿大、撒哈拉沙漠以南非洲的 IP 地址信息；欧洲 IP 地址注册中心 RIPE（Reséaux IP Européens）包括了欧洲、北非、西亚地区的 IP 地址信息；亚太地区网络信息中心 APNIC（Asia Pacific Network Information Center）包括了东亚、南亚、大洋洲 IP 地址注册信息；拉丁美洲及加勒比互联网络信息中心 LACNIC（Latin American and Caribbean Network Information Center）包括了拉丁美洲及加勒比海诸岛 IP 地址信息。

中国的 IP 地址管理机构称为中国互联网络信息中心（China Internet Network Information Center，简称 CNNIC），它是成立于 1997 年 6 月的非营利管理与服务机构，行使国家互联网络信息中心的职责。中国科学院计算机网络信息中心承担 CNNIC 的运行和管理工作。它的主要职责包括域名注册管理，IP 地址、AS 号分配与管理，目录数据库服务，互联网寻址技术研发，互联网调查与相关信息服务，国际交流与政策调研，承担中国互联网协会政策与资源工作委员会秘书处的工作。

1.3 Internet 协议与标准

Internet 的实质是实现异种网络的互联，它充分利用各种通信子网的数据传输能力，通过在依赖于通信子网的通信模块和应用程序之间插入新的协议软件来保证应用程序之间的互操作性。因特网的协议族称为 TCP/IP 协议族。其中包含了为数众多的协议，如应用层的 Telnet、FTP、HTTP、SMTP、DNS 等协议、传输层 TCP、UDP 协议，网络层的 IP、ARP、

RARP、ICMP、IGMP 等协议。

Internet 的一个公认标准是 RFC（Request For Comment），RFC 可以说是 TCP/IP 和 Internet 发展及成长的基石，所有关于 TCP/IP 和因特网的规格、协议内容、会议记录、发展历史等文件数据都以 RFC 数字编号的方式，由美国网络信息中心（Network Information Center，NIC）所收集。例如 RFC1000 介绍了一些 RFC 的历史，以及各种 RFC 的分类。若有人对于改进 TCP/IP 现有能力有新的想法时，可以写一个计划方案发表在 Internet 上，这个计划方案即是所谓的 RFC。RFC 的作者都是自愿的，其创作得不到任何报偿。每个 RFC 会被赋予一个号码，此号码为一递增的数字，绝不会被重新指定。更新的 RFC 有更高的数字编号，并使得旧的 RFC 失效，因此若发现在不同的文件中讨论的是相同的主题，应以编号较高的 RFC 为依据。另外，亦可能有自愿的评论者对 RFC 作建设性的批评与建议，原作者可根据以校订原先的设计使之更加完美。若一切无问题，该项 RFC 便成为起草标准（Draft Standard），程序设计人员就可依该份标准来设计软件，实现其所描述的功能。在真正的程序代码出现之前，RFC 都不被认定是正式标准。

1.4 Internet 应用现状与发展趋势

从目前的情况来看，Internet 市场仍具有巨大的发展潜力，未来其应用将涵盖从办公室 共享信息到市场营销、服务等广泛领域。另外，Internet 带来的电子贸易正改变着现今商业活动的传统模式，其提供的方便而广泛的互联必将对未来社会生活的各个方面带来影响。

然而 Internet 也有其固有的缺点，如接入网络缺乏整体规划和设计，网络拓扑结构不清晰以及容错及可靠性能的缺乏，而这些对于商业领域的不少应用是至关重要的。安全性问题是困扰 Internet 用户发展的另一主要因素。虽然现在已有不少的方案和协议来确保 Internet 网上的联机商业交易的可靠进行，但真正适用并将主宰市场的技术和产品目前尚不明确。另外，Internet 是一个中心的网络。所有这些问题都在一定程度上阻碍了 Internet 的发展，只有解决了这些问题，Internet 才能更好地发展。

随着世界各国信息高速公路计划的实施，Internet 主干网的通信速度将大幅度提高；有线、无线等多种通信方式将更加广泛、有效地融为一体；Internet 的商业化应用将大量增加，商业应用的范围也将不断扩大；Internet 的覆盖范围、用户入网数以令人难以置信的速度发展；Internet 的管理与技术将进一步规范化，其使用规范和相应的法律规范正逐步健全和完善；网络技术不断发展，用户界面更加友好；各种令人耳目一新的使用方法不断推出，最新的发展包括实时图像和话音的传输；网络资源急剧膨胀。总之，人类社会必将更加依赖 Internet，人们的生活方式将因此而发生根本的改变。

第 2 章　TCP/IP 协议族体系结构

TCP/IP 协议族无疑是当今流行最为广泛的网络互联协议，人们今天所熟悉的绝大多数 Internet 服务都是架构在该协议族之上的，对于它的历史已在第 1 章中有所了解。那么它究竟是如何将运行不同操作系统，由不同厂家生产的计算机互联起来的？它和已有的 OSI 模型之间的关系如何？它的每一层具体包含哪些协议？我们将在本章中一一阐述。

2.1　TCP/IP 层次结构及其与 OSI 七层体系结构的比较

TCP/IP 与 OSI 的体系结构都是采用分层结构，结构中的下层向上层提供服务。这种分层结构具有模块划分清晰，扩展性好等优点，所以被 TCP/IP 和 OSI 所采用。虽然 TCP/IP 和 OSI 都是采用分层结构，它们之间还是存在着许多重要的区别。

2.1.1　分层体系结构的对应

TCP/IP 与 OSI 分层架构间的对应，可以用图 2.1 来表示。OSI 具有完整的七层架构，而 TCP/IP 则只定义了 3 种层次的服务。TCP/IP 应用服务层，对应到 OSI 架构中的应用层、表示层以及会话层。两者之间最大的不同点在于：OSI 考虑到开放式系统互联而设定了数据表示层，而 TCP/IP 的网络层与传输层，则分别与 OSI 的网络层和传输层的功能大致相同。此外，TCP/IP 本身并没有提供物理层与数据链路层的服务，所以一般是架在 OSI 的第一、二层上运作。

OSI	TCP/IP
应用层	应用层
表示层	
会话层	
传输层	传输层
网络层	Internet层
数据链路层	网络接口层
物理层	

图 2.1　TCP/IP 与 OSI 分层结构的对应

2.1.2　总体发展

TCP/IP 的发展比 OSI 早了约 10 年左右，技术上的发展较成熟，开发出来的相关应用协

议也较多，此外，由于它是应因特网的实际需求而产生的，因此在现实的环境中可行性也较高。而 OSI 架构完整、功能详尽、包容性大，但在 Internet 中大多数还属于测试阶段，很少有实际运行的系统。

就目前的发展状况来说，TCP/IP 已成为了 Internet 中的主流协议，在使用上比 OSI 要广泛许多。它具有非常多的应用标准，对于现行网络应用系统的开发而言，能提供较多的规划选择，而且由于 TCP/IP 已在实际中使用相当长的时间，具有此方面开发与使用经验的人员也比较多。

2.1.3　标准及规范

TCP/IP 产生于 Internet 的研究和实践中，是应实际需求而产生的实作，本身在发展之前并没有事先定义一个严谨的架构。而 OSI 则是由标准化组织所制定，先定义了一个功能完整的架构，再根据该架构发展相应的协议。

TCP/IP 虽然有 IAB、IETF 等机构负责制定与讨论 TCP/IP 的标准化，也有许多学术界人士和计算机厂商参与，但是却没有一个正式的单位负责测试验证厂商所开发的 TCP/IP 通信软件是否完全遵照标准的规范设计，所以对使用者而言，惟一的保障是借着各系统之间的互连互通测试经验，以确实证明其所使用的网络系统是否可以与其他系统上的 TCP/IP 网络应用功能互通。而 OSI 则有专门的单位来进行规范性测试以及互通性测试，这一点对使用者而言，是一个很重要的保障，但是由于测试通常需花费一段不短的时间（约需二至三年），所以一般效率不高，经常出现的情况是标准已经出来，但是却在市场上找不到可用产品，与市场需求不太相符。

2.1.4　网络层

TCP/IP 的网络层与 OSI 架构中的网络层的功能大致相仿。若以实际协议来做比较，TCP/IP 的 IP 与 OSI 的 CLNP（Connectionless Network Protocol），其主要差别在于寻址方式的不同。TCP/IP 将网络上每一点的地址定为 32 位的固定长度，TCP/IP 网络上的每一个系统都至少具有一个惟一的地址与其他系统通信，但对于同时提供两个网络接口连接不同网络的系统（如网关）而言，则必须拥有两个以上的地址，这在网络地址管理及对网络其他点的通信进行上，则显得较为麻烦。而且以长远的角度来看，现有的寻址方式将不能容纳网络上愈来愈多新增的系统，因此，目前许多人员正在研究 IPv6 这一新协议，以满足未来不断扩展的需求。

OSI 所定的地址空间为不固定的可变长，必须由所选定的地址命名方式（Authority and Identifier）决定，最长可达 160 位（20 bytes）。依照 OSI 中有关地址标准的规范，网络上每一个系统至多可有 256 个通信地址，而且因为 OSI 所定义的网络地址与网络接口无关，所以网络地址的安排将不受限于网络接口。由于其地址长度较长，因此将可容纳网络上更多的系统，具有较大的增长空间。

2.1.5　传输层

TCP/IP 在传输层中有 TCP 与 UDP 两种协议，各具有面向连接与无连接的性质。OSI 在



制定传输层的标准时，主要是参考 TCP/IP 协议族，而定义了五个等级的不同层次服务。其中 TCP/IP 的 TCP 与 OSI 的 TP4、TCP/IP 的 UDP 与 OSI 的 TP0 的架构及功能大体上是相同的，只是其内部细节有一些差异。

2.1.6 应用层

应用层的功能应该是面向最终用户的，因此它们是千差万别的。常见的应用有以下几种：

2.1.6.1 远程登录

TCP/IP 的远程登录标准为 TELNET，OSI 所定的远程登录标准称为虚拟终端 VT（Virtual Terminal）。由于不同的终端机会有各种型号，因此在 TCP/IP 的 TELNET 与 OSI VT 虚拟终端标准中都提供了协商（Negotiation）机制，通信两端在通信之前，会进行协商，并交换终端机环境参数（Profile），直到彼此达到共识之后才进行应用系统与客户端之间的数据交换。

除了共有的协商功能之外，OSI VT 所提供的终端机参数模型比 TELNET 多。

（1）VT 能支持控制的转换，例如 ASCII 的警铃字符（BELL）与 EBCDIC 的警铃字符各有不同的内码定义。而 TELNET 则无法处理这类因字符集不同所造成的字符功能差异。

（2）VT 能够提供更多的字符显示属性，例如在终端屏幕上每一个字符都可以定义其颜色及字型等功能。

（3）VT 能够提供一维、二维、三维显示区域。而 TELNET 只提供翻页显示模式（Scroll Mode）。

2.1.6.2 文件传输

以 TCP/IP 所制定的 FTP 与 OSI 所制定的 FTAM（File Transfer, Access and Management）而言，两者都提供了基本文件处理功能，如文件复制、删除、目录查询、更改文件名、文件属性的对应等。但是由于 TCP/IP 中并未定义数据表示层来执行不同系统间数据储存内码转换的功能，因此，虽然 FTP 可传送任何一种数据类型文件（如 IMAGE、ASCII、EBCDIC 等），但是却不能在几种不同的文件格式之间自动地进行内码转换处理。OSI 在 FTAM 中则定义了一套虚拟文件储存器（Virtual File Store），使得不同计算机系统间在交换文件时，首先将本身的文件格式与文件属性对应转换成标准的虚拟文件储存格式送出，对方在收到此虚拟文件数据后，再根据其自己的文件系统与虚拟文件储存的对应关系，转换成属于自己的文件格式，以解决不同文件系统间对文件格式处理方式不尽相同的问题。

以文件传输效率而言，由于 OSI FTAM 在文件传输的过程中，必须先将文件对应成虚拟文件格式，再通过表示层的抽象符号语法转换成网络上标准的文件传输数据，会花费许多时间。反观 TCP/IP FTP，不但省去了上述的转换动作，同时还可依文件的性质，选择压缩模式（Compressed Mode），提供更有效率的文件传输方法。因此平均而言，TCP/IP FTP 将会比 OSI FTAM 文件传输的效率高。

TCP/IP FTP 与 OSI FTAM 在不同的功能层次虽各擅胜场，但就目前的使用状况来说，TCP/IP FTP 却比 OSI FTAM 要普遍得多。

2.1.6.3 邮件传输

在邮件处理方面，由于 TCP/IP 的 SMTP 发展时间较早，因此目前采用得较广泛。但 OSI

所制定的 MHS（Message Handling Systems），在功能上比 TCP/IP SMTP 更完整，且具有传输过程可靠、邮件处理的功能较多等应用潜力。以下列出 OSI MHS 所提供的一些 TCP/IP SMTP 没有的功能：

（1）可指定邮件传递的时间与邮件的处理优先级（如急件、速件、正常邮件），并设定邮件内容的敏感度（Sensitivity）。

（2）可设定双挂号邮件传递以确认邮件已正确送抵收件人邮箱中，或传回递送失败报告给发信人查明失败原因。

（3）邮件的内容格式除了文字以外，尚能传输许多其他的多媒体类型。而在 TCP/IP SMTP 中，文件的内容格式只能是一般的文字文件，若要传输多媒体的数据则需配合 MIME 协议。

2.1.6.4　**网络管理**

网络管理方面，TCP/IP 与 OSI 的协议均不限定只能在本身的网络上运作：TCP/IP 所发展出来的网络管理协议可以利用其他通信网路协议作运行的基础，而 OSI 的相应协议也可以在 TCP/IP 的通信网路协议之上运行(称为 Common Management Information Service From OSI on TCP/IP, CMOT）。因此，我们将范围限定在 SNMP 与 CMIP 两者间的本身功能作比较。

基本上，SNMP 所提供的是一套能满足基本需求的简单网络管理协议，它具有简单、容易制作等优点，执行时所占用的内存及使用的 CPU 资源也较少。由于其实际被利用的经　验较多，目前在网络上使用的广泛程度大于 CMIP。以下将 SNMP 与 CMIP 功能的异同整理如下。

SNMP 与 CMIP 都提供以下的管理功能。

（1）网络发生错误时的处理（Fault Management）。

（2）网络运行性能评估（Performance Management）。

（3）网络会计管理（Accounting Management）。

（4）被管理对象（Managed Object）的名称配置与鉴别（Configuration and Name Management）。

不同之处在于：

（1）通信联机：CMIP 会预先建立通信管道才执行管理命令，而 SNMP 则采用无连接方式的网络联机管理方式，所以平均而言 SNMP 的额外负担较少，但相对地，CMIP 通信管理质量则较为稳定可靠。

（2）管理模式：SNMP 采用轮询式（Polling-based）的管理模式，管理者会定期询问被管理者，两者间的联系较密切，不过这种方式会使网络上的数据传输量增加，同时在网络上同一个管理者将无法同时管理网络上的许多点。CMIP 采用基于事件的管理模式(Event-based Management），被管理者利用异步的方式，将预定发生的事件通知管理者处理。

（3)管理信息:SNMP 与 CMIP 的网络信息都是使用对象(Object)来表示,并采用　ISO 所定义的抽象符号语法描述基本编码方式（BER-Basic Encoding Rule）来表示一个对象。但目前 SNMP 仅使用了部分的抽象符号语法规则，因此所能定义的对象类型及属性比 CMIP 所能定义的少。

（4）网络安全：CMIP 的在网络安全的管理功能比 SNMP 完整，可提供管理之间的鉴

权（Authentication）、访问控制（Access Control）、加密密钥管理（Key for Enciphering Code）、授权（Authorization）、安全日志（Security Log）等安全管理机制。而 SNMP 在这方面的机能上相对较弱。

就目前的网络环境而言，真正在 TCP/IP 中发展的应用环境比较成熟，且它在运行时所需要的系统资源较少。除此之外，TCP/IP 还提供了网络文件系统（NFS）、远程调用（Remote Procedure Call，RPC）、窗口操作系统（X-Windows）等其他应用背景，而这些功能则尚未在 OSI 定义的应用标准中。所以，目前有一些应用需求无法利用 OSI 的方式来设计。

总体而言，OSI 所制定的应用标准，大致上来说比 TCP/IP 的相应功能完整丰富且精细，但却始终无法在电信网络及数据网络中快速成长。除了上述的原因之外，可能也是由于其相关产品过于复杂，且需要庞大的人力与经费支持。此外，OSI 的标准制定过程太过缓慢，也使它的应用受到限制。

上面已经从宏观上了解了 TCP/IP 协议栈的架构，并比较了其和 OSI 七层模型的异同，那么作为核心的 IP 层究竟是如何工作的呢？运行在网络层上的路由器如何互联异种网络？下面，深入探讨一下其工作原理。

2.2 路　由　器

路由器作为网络层的核心设备，在网络中处于至关重要的位置。它可以连接不同类型的网络，能够选择数据传输路径并对数据进行转发。从通信角度看，路由器仍然是一种中继系统，与中继器和网桥类似，但从计算机网络的角度看，它与中继器和网桥设备是运行在协议栈不同层之上的。在 IP 网络发展的最初，早期路由器就被称为网关，这主要是因为它们常常作为本地校园网和广域网之间的网络接口。现在的网关则是指网络层以上的中继系统，如应用层网关。但是，由于习惯问题，一些固有的称呼仍然被保留，因此往往需要根据上下文意思判断其具体含义。

路由器是用于连接多个逻辑上分开的网络，因此，路由器具有转发报文和路由选择两大功能，它能在异种网络互联环境中，建立灵活的连接，可用完全不同的数据分组和介质访问方法连接各种子网。它不关心各子网使用的硬件设备，但要求运行与网络层协议相一致的软件。一般来说，异种网络互联与多个子网互联都应采用路由器来完成。

2.2.1 路由器的工作原理

当 IP 子网中的一台主机发送 IP 报文给同一子网的另一台主机时，它将直接把 IP 报文送到网络上，对方就能收到。而要送给不同 IP 子网上的主机时，它要选择一个能到达目的子网上的路由器，把 IP 报文送给该路由器，由路由器负责把 IP 报文送到目的地。如果没有找到这样的路由器，主机就把 IP 报文送给一个称为“默认网关”（default gateway）的路由器上。“默认网关”是每台主机上的一个配置参数，它是接在同一个网络上的某个路由器端口的 IP 地址。

路由器转发 IP 报文时，只根据 IP 报文目的 IP 地址的网络号部分，选择合适的端口，把 IP 报文送出去。同主机一样，路由器也要判定端口所接的是否是目的子网，如果是，就直接

把报文通过端口送到网络上，否则，也要选择下一个路由器来传送报文。路由器也有它的“默认网关”，用来传送不知道往哪儿送的 IP 报文。这样，通过路由器把知道如何传送的 IP 报文正确转发出去，不知道的 IP 报文送给“默认网关”路由器，这样一级级地传送，IP 报文最终将送到目的地，送不到目的地的 IP 报文则被网络丢弃了。

2.2.2　路由器的功能

前面提到路由器具有转发报文和路由选择两大功能，那么它们各自是如何工作的呢？转发的意思是指为经过路由器的每个数据报文寻找一条最佳传输路径，并将该数据有效地传送到目的站点。路由器首先在路由表中查找，判明是否知道如何将分组发送到下一个站点（路由器或主机），如果路由器不知道如何发送分组，通常将该分组丢弃；否则就根据路由表的相应表项将分组发送到下一个站点，如果目的网络直接与路由器相连，路由器就把分组直接送到相应的端口上。这就是路由转发协议（routed protocol）。下面以作者本机中的路由表为例来说明转发的过程（注：本机的 IP 地址为 211.65.59.36）。

在命令提示符下运行 netstat –rn，将出现如下信息：

```
Route Table
======================================================================
Interface List
0x1 ........................... MS TCP Loopback interface
0x2 ...00 03 0f fe 3a ec ...... DigitalChina DCN-530TX Fast Ethernet Adapter
   -
数据包计划程序微型接口
======================================================================
======================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
          0.0.0.0          0.0.0.0      211.65.59.4    211.65.59.36     20
        127.0.0.0        255.0.0.0        127.0.0.1       127.0.0.1      1
       172.16.0.0      255.240.0.0      211.65.59.1    211.65.59.36      1
      202.119.0.0    255.255.224.0      211.65.59.1    211.65.59.36      1
    202.119.144.0    255.255.240.0      211.65.59.1    211.65.59.36      1
      211.65.32.0    255.255.224.0      211.65.59.1    211.65.59.36      1
      211.65.59.0    255.255.255.0     211.65.59.36    211.65.59.36     20
     211.65.59.36  255.255.255.255        127.0.0.1       127.0.0.1     20
    211.65.59.255  255.255.255.255     211.65.59.36    211.65.59.36     20
        224.0.0.0        240.0.0.0     211.65.59.36    211.65.59.36     20
  255.255.255.255  255.255.255.255     211.65.59.36    211.65.59.36      1
Default Gateway:       211.65.59.4

======================================================================
Persistent Routes:
  Network Address          Netmask  Gateway Address       Metric
      211.65.59.0    255.255.255.0     211.65.59.36            1
      202.119.0.0    255.255.224.0      211.65.59.1            1
    202.119.144.0    255.255.240.0      211.65.59.1            1
```

```
211.65.32.0      255.255.224.0      211.65.59.1        1
 172.16.0.0        255.240.0.0      211.65.59.1        1
```

第 1 行说明，如果报文目的地址是 loopback 地址 127.0.0.0，将通过 loopback 接口 127.0.0.1 送达。这里简要说明一下网络 127.0.0.0 作为 IP 通路被保留到主机。通常，地址 127.0.0.1 将在主机上被分到一个特殊的接口，即所谓的 loopback 接口，它像一个关上的电路一样行动。来自 TCP 或 UDP 被传递给它的任何 IP 包将被返回到它们自身，好像它刚从一些网络到达了。这允许用户开发并且测试曾经没有使用一个“真实”网络的联网软件。

第 4 行说明，如果报文目标网络地址是 202.119.0.0，则将通过作者所在 LAN 的网关 211.65.59.1 转发至目标子网。

第 8 行说明，报文的目标地址就是本机的话，则直接通过 loopback 接口 127.0.0.1 发送到本机。

Default Gateway 说明：当搜索完路由表中的所有表项都不能找到匹配待发报文目标地址，则将待发报文转往 Default Gateway。

路由选择即判定和构造到达目的地的最佳路径，由路由选择算法来实现。由于涉及到不同的路由选择协议和路由选择算法，要相对复杂一些。为了判定最佳路径，路由选择算法必须启动并维护包含路由信息的路由表，其中路由信息依赖于所用的路由选择算法而不尽相同。路由选择算法将收集到的不同信息填入路由表中，根据路由表可将目标网络与下一跳（nexthop）的关系告诉路由器。路由器间互通信息进行路由更新，更新维护路由表使之正确反映网络的拓扑变化，并由路由器根据量度来决定最佳路径。这就是路由协议（routing protocol），例如路由信息协议（RIP）、开放式最短路径优先协议（OSPF）和边界网关协议（BGP）等，我们将在第 6 章给予详细阐述。

简言之，路由转发协议可视为一个如何查找路由表以决定最佳传输路径的一组规则，而路由协议则是根据路由选择算法搜集到的网络信息构造路由表表项的方法。

2.3　TCP/IP 各层协议组成

TCP/IP 各层协议的详细描述将在后文中予以介绍，这里先从宏观上认识一下 TCP/IP 协议栈的架构并简单介绍各层协议的功能。由图 2.2 可了解 TCP/IP 协议在整个协议族所占位置的重要性。

应用层	Telnet	FTP	HTTP	SMTP	DNS
传输层	TCP		UDP		
网络层	ARP	RARP	ICMP	IGMP	IP
数据链路层	逻辑链路子层				
	介质访问子层				
物理层	SONET/SDH/PDH				

图 2.2　TCP/IP 协议集族

从图 2.2 中可以看到，Internet 协议族主要功能集中在第 3～4 层，通过增加软件模块来保证和已有系统的最大兼容性。

在网络层上，地址解答协议 ARP（Address Resolution Protocol）实现 IP 地址向物理地址的映射；反向地址解答协议 RARP（Reverse Address Resolution Protocol）实现物理地址向 IP 地址的映射；网络层协议 IP 提供结点之间的报文传送服务；Internet 报文控制协议 ICMP（Internet Control Message Protocol）传输差错控制信息以及主机/路由器之间的控制信息；Internet 组管理协议 IGMP（Internet Group Management Protocol）能让一个物理网络上的所有系统知道主机当前所在的多播组。

在传输层上，传输控制协议 TCP（Transmission Control Protocol）提供用户之间的面向连接可靠报文传输服务；用户数据报协议 UDP（User Datagram Protocol）提供用户之间的不可靠无连接的报文传输服务。

在应用层上，Telnet 提供远程登录（终端仿真）服务；文件传输协议 FTP（File Transfer Protocol）提供应用级的文件传输服务；简单邮件传输协议 SMTP（Simple Mail Transfer Protocol）提供简单的电子邮件发送服务；超文本传输协议 HTTP（HyperText Transfer Protocol）提供万维网浏览服务；域名系统 DNS（Domain Name System）负责域名和 IP 地址的映射；其他服务（Others）支持其他的应用服务。

因特网通过信息的传输向用户提供各种服务，类似 OSI/RM，各层可提供的服务通过各层的控制协议，并经过各层实体的合作予以实现。

第 3 章　IP　协　议

IP 协议是整个 TCP/IP 协议族中最重要的协议。它位于物理链路层之上，向上层协议屏蔽了各种不同的物理链路的差别，因此能将各种不同介质的网络互联起来。所有在 Internet 上传输的数据都以 IP 数据包格式传输。IP 提供不可靠（unreliable）、无连接的数据包传送服务，即它不能保证 IP 数据包能成功地到达目的地。IP 仅提供最好（best-effort）的传输服务。

在本章中，我们将主要介绍 IP 协议的工作原理及 IP 数据包的详细格式，但 IP 数据包中与错误控制等相关的内容将放在相应章节进行介绍。

3.1　IP 协议的目的与工作原理

IP 协议的目的是提供不可靠（unreliable）、无连接的数据包传送服务，仅提供最好的（best-effort）传输服务。因此可以看出 IP 协议就是要将 Internet 上的数据尽可能地从产生地传输到数据的目的地。为了了解 IP 协议的详细的工作机制，必须先了解 IP 协议的工作原理及其中包含的一些基本概念，因此本节将向读者提供这些知识。

3.1.1　IP 协议数据的传输过程

从第 2 章可以知道，整个 TCP/IP 协议族的工作都是以数据包和存储转发机制为基础的，IP 协议也使用这一原理。下面，以一个形象的比喻来说明 IP 协议数据的传输过程。

我们先看看传统的邮件传递系统的工作过程，如图 3.1 所示。A 市 A 区的发送人王某要写一封信件给 B 市 B 区的李某。那么在王开始写信到李接收到信件这个过程中都发生了一些什么事情呢？首先王要将写好的信装进一个信封，并在信封上写好接收人的地址（例如北京市清华大学）和接收人姓名，然后将信投进邮局 A 设的信箱。邮局 A 的邮递员将信箱中的信件带回邮局 A。所有的信件在邮局 A 根据其目的地进行分类并打包。打包后的信件会运算到 A 市的总邮局 A。在总邮局 A，信件包将被运送到 B 市的总邮局，这次运送可以通过多种交通方式，如汽车、火车、飞机等。总邮局将根据信件接收人的详细地址将信件分发到不同的区邮局，那么李的信件就运送到了邮局 B。最后邮局 B 的邮递员将信件投递到李的信箱，李就可以从信箱接收到邮件了。

我们可以将 IP 协议的工作过程和信件递送的过程进行比较，它们之间是非常类似的。我们可以将图 3.1 中的区看作是主机系统（工作站、服务器等），将市看作是局域网，而将整个邮政系统看作是 Internet。在发送方，主机系统首先将要发送的数据准备好并指明目的地，然后将数据交给 IP 协议（即邮局 A）。IP 协议将数据组成 IP 包（即将信件打包）并将数据包发送到网络设备（交换机或路由器），由网络设备将数据包传送到正确的目的地。在接收方，网络设备接收到 IP 数据包后会根据其目的地址将数据包发送给主机系统。这样整个数据

的传送就完成了。

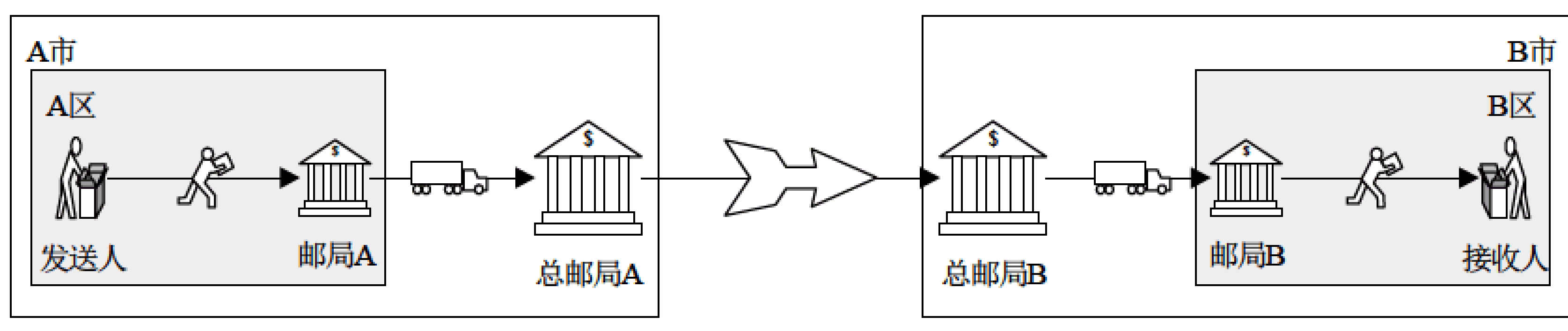


图 3.1 邮政系统信件传递过程

虽然 IP 协议的工作原理与信件递送过程相似，但也存在着一定的区别：

（1）信件递送过程中，信件的收集是定时的。图 3.1 中邮局 A 的邮递员会在每日定时前往邮箱收集信件。但在 IP 协议中，只要待发送的数据准备好，随时可以交给 IP 协议处理。

（2）信件打包时，只会组包，即将多封信件打成一个包。在 IP 协议中不会将小的数据组成一个大包，只会将过大的数据分解成小包。

（3）在图 3.1 中，A 市发往 B 市不同区的包可能会打在同一个包中，然后信件包在总邮局 B 需要按不同区进行分解和重新组合。在 IP 协议中不会将不同地址的数据包组合成一个包。

3.1.2 IP 协议中的概念

IP 协议的工作过程中包含了几个重要概念。

首先，IP 协议的工作对象是数据包。IP 协议在处理数据包时，只会根据数据包本身的情况来处理数据包，不会考虑不同数据包之间的关系。因此，对两个目的地相同的数据包的处理可能是不同的。在介绍 IP 协议数据包格式时我们将详细说明 IP 协议是如何处理数据包的。

其次，IP 协议的责任是将数据正确传递到目的地。那么 IP 协议必须能够表示和识别数据包的地址，并能根据地址选择数据包传递的路径。其中第一个问题将在 3.2 节说明。而路径选择问题放在第 6 章介绍。

最后，IP 协议是提供不可靠数据传递服务的协议。正如信件有时无法投递一样，IP 协议的数据包有时也无法传送到目的地。在 IP 协议中有一整套的错误处理机制用来诊断并处理数据包无法传送的情况。

3.2 IP 地 址

正如上节所述，IP 协议要将数据正确传递到目的地，就必须能够表示和识别数据包的地址。IP 地址是一种用二进制数表示的地址，它是 Internet 上主机的惟一标识。Internet 上的每一台主机都被赋予一个惟一的 32 位的整数，这个整数就是 IP 地址。为了便于根据 IP 地址寻找到代表该地址代表的主机，这个整数被分为两个部分：网络 ID（netid）和主机 ID（hostid）。由于 IP 数据包的传递是根据 IP 地址进行的，因此 IP 地址结构和组织的合理性就会影响数据包传递的效率。为此，IP 地址就包含了一些复杂的技术，我们在本节中加以详细说明。

3.2.1　IP 地址的分类

IP 地址分为 A、B、C、D、E 五类，如图 3.2 所示。

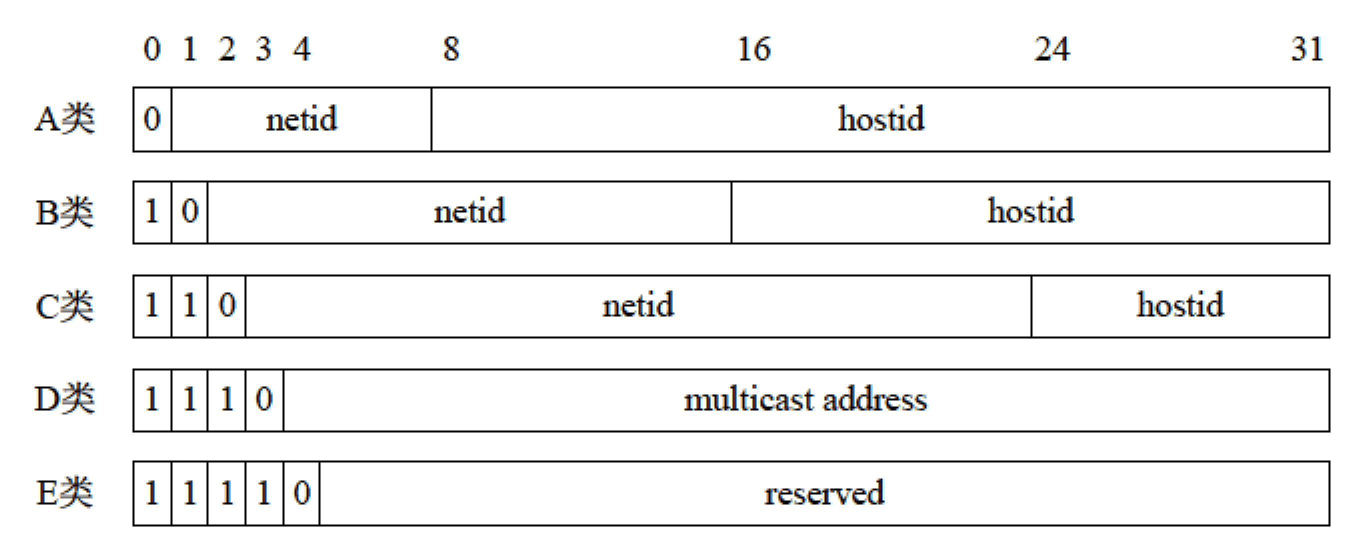


图 3.2　IP 地址分类

从图 3.2 可以看出，五类 IP 地址中，目前使用其中的前四类，且 D 类地址是多播地址。那么用来标识网络上的主机和设备的就只有前面的三类了。A、B、C 三类 IP 地址都由三部分组成：地址类别标识、netid 和 hostid。其中地址类别标识表明了该地址所属的地址类别。netid 则是某个网络的标识，所有 netid 部分相同的 IP 地址都属于同一个网络。hostid 则是用来区分同一个网络中的不同主机和设备的。同传统的通信地址相比，netid 就类似于市名，hostid 则是分邮局的管理范围，至于收信人的确切地址，我们在讨论 UDP 和 TCP 时再进行对比。由于 IP 地址有上述的结构，所以网络设备在转发 IP 数据包时，只需要先根据 IP 地址的 netid 找到该 IP 地址所属的网络，再由该网络中的设备根据 hostid 将数据包转发给网络中的相应主机或设备。这与传统邮局投递系统的原理是相同的。

我们再来看看前三类地址的格式。为什么要将它们分为 A、B、C 三类呢。从图 3.2 中可以看到，A 类地址的 netid 部分短而 hostid 部分长，而 C 类地址的 netid 部分长而 hostid 部分短。因此 A 类地址能表示的网络少，但单个网络中的主机或设备可以很多，而 C 类地址能表示的网络多，但单个网络中的主机和设备较少，B 类地址的各项都适中。之所以这样分是为了节省 IP 地址资源，使得不同规模的网络可以使用相应类别的 IP 地址，不至于造成 IP 地址的浪费。例如大公司的网络是一个大型网络，那么它就可以使用 A 类地址。小公司的网络中的主机和设备比较少，它就可以使用 C 类地址。由于 Internet 的迅速发展，使得接入的主机和设备飞速增多，造成了 IP 地址资源的紧张。我们在后面还会讨论其他的 IP 地址技术，它们可以更有效地利用 IP 地址资源。

我们还需要对前三类 IP 地址的 hostid 进一步说明。通常所有位都为 0 和所有位都为 1 的 IP 地址被赋予了特殊的意义，不能用来表示网络中的主机和设备。所有位都为 0 的 IP 地址用来代表整个网络本身，而所有位都为 1 的 IP 地址则用来表示网内广播。

对 netid 也有一个特殊情况，就是所有位都为 0 的 netid。这样的 netid 用来代表本网络。这对于那些需要进行通信却不知道本网络地址的主机非常有用。

3.2.2　IP 地址的表示

由于 IP 地址是一个 32 位的整数，所以很自然地，在 IP 协议的实现中，IP 地址就被表示

为一个 32 位的整形数。但这种表示方法不方便人的阅读与记忆。所以 IP 就被显示为四个由点隔开的十进制整数。即在图 3.2 中按照从左到右的顺序将 32 位分为 4 个 8 位，每个 8 位的十进制表示为一个部分。如 IP 地址：

11001010 01110111 00001001 00011110

就表示为：

202.119.9.30

这种表示方法便于人的阅读、记忆和交流。表 3.1 是五类 IP 地址的地址范围。

表3.1 五类IP地址的地址范围

IP 地址类别	最小地址	最大地址
A	0.1.0.0	126.0.0.0
B	128.0.0.0	191.255.0.0
C	192.0.1.0	223.255.255.0
D	224.0.0.0	239.255.255.255
E	240.0.0.0	247.255.255.255

在表 3.1 中，我们只列出了 A、B、C 三类地址中的网络地址。从表 3.1 中可以看出，A、B、C 三类地址没有包含所有可能的地址。例如：A 类地址 127.0.0.0 没有列出。这是因为地址 127.x.x.x 被保留用作回环（loopback）地址。目标地址为 127.x.x.x 的数据包是不会发送到网络上的，而是返回到本机，就好像是接收到一个数据包一样。回环地址的目的是为了方便 TCP/IP 的测试及进程间通信。

3.2.3 特殊 IP 地址总结

根据对 IP 地址的描述，我们可以对一些具有特殊作用的 IP 地址进行简单归纳，如图 3.3 所示。

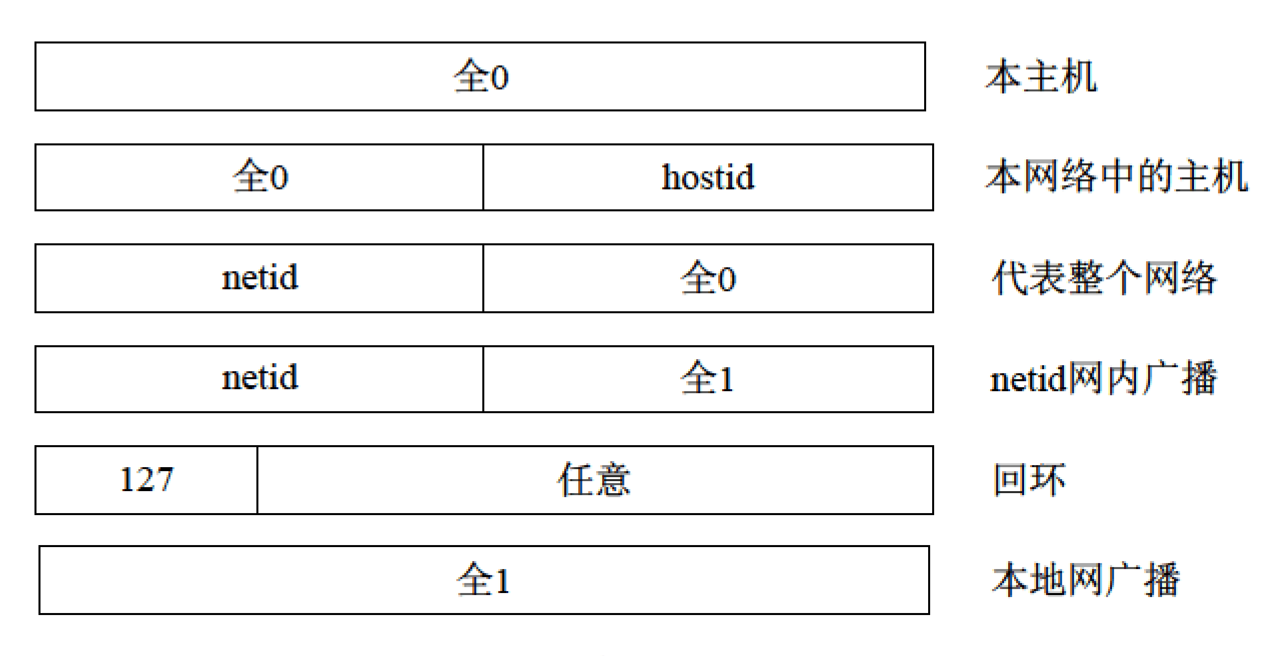


图 3.3 特殊用途 IP 地址

在图 3.3 中只列出了 netid 和 hostid，不包含地址类别标识。

3.2.4 IP 地址的缺陷

IP 地址是一种结构性很强的地址，其原理类似于现实世界的地址表示方式，便于 IP 数

据包的路由转发。但 IP 地址也存在自己的缺陷。

首先，IP 地址标识的网络主机或设备与 Internet 的连接关系，而不是标识主机和设备本身。所以当主机从一个位置移动到另一个位置时，它原有的 IP 地址将会变得无法使用，需要重新配置一个新的 IP 地址，这个新 IP 地址是属于它现在所处网络的地址。

其次，不同类 IP 地址之间巨大的差异已经浪费了大量的地址。举例来说，一个中等规模的公司需要 300 个 IP 地址。一个 C 类地址（254 个地址）不够用。使用两个 C 类地址，提供的地址有富余，但是这样一来，一个公司就有两个不同的网络，增加了路由表的尺寸——每一个地址空间需要一个路由表项（即使它们属于同一个组织）。另一种选择是，B 类地址提供了所有需要的地址，而且是在一个网络中。但是这样却浪费了 65234 个地址，当一个网络有多于 254 个主机时就提供一个 B 类地址，这种情况太常见了。因此，B 类地址比其他地址更容易耗尽。

再次，当一个组织的 C 类网络扩大时，它需要更改整个网络中所有主机与设备的 IP 地址，相应的需要对各种软件系统的配置进行修改，很容易出错。

但这些问题都得到了解决。下面开始讨论 IP 地址的各种扩展技术。

3.2.5 子网技术

对于某些中等规模的网络，一个 C 类的 IP 地址不够用，而使用 B 类的 IP 地址则过于浪费。如果使用多个 C 类地址则会造成一个组织有多个网络域，这会导致路由器中的路由表会扩大。这个问题在 Internet 发展的初期并不突出，但随着 Internet 的飞速发展，接入网络的增多，问题就逐渐突出了。解决方法是分层地组织这些同一组织的不同网络，并在它们之间路由。从 Internet 的角度看具有多个网络的组织应该被看作只有一个网络。因此，它们应该共享一个共同的 IP 地址范围。

所谓的子网技术就是将原有的两层结构（netid 和 hostid）的 IP 地址分为三层：netid、subnetid 和 hostid。subnetid 和 hostid 是由原先 IP 地址的 hostid 部分分割成两部分得到。因此，用户分子网的能力依赖于被子网化的 IP 地址类型。IP 地址中 hostid 位数越多，就能分得更多的子网和主机。然而，子网减少了能被寻址主机的数量。实际上，是把主机地址的一部分拿走用于识别 subnetid。子网由子网掩码标识。

子网掩码是 32 位二进制数，可用与 IP 地址标识格式一样的点-十进制数格式标识。子网掩码告诉网络中的端系统 IP 地址的多少位用于识别网络和子网。这些位被称为扩展的网络前缀。剩下的位标识子网内的主机，掩码中用于标识网络号的位置为 1，主机位置为 0。

虽然使用子网掩码可以将一个网络分为多个子网，但它还是有缺陷的，即一个网络只能支持一个子网掩码。也就是说，一旦确定了网络的子网掩码，那么网络中各子网所允许的子网规模都是一样的。为了克服这个缺点，VLSM（Variable-Length Subnet Mask）技术就应运而生。VLSM 允许一个网络使用不同的网络掩码以适应不同规模子网的要求，它使一个组织的 IP 地址空间被更有效的使用。

为了说明 VLSM 的原理与作用，我们举一个例子。假设有一个组织的网络规模可以使用一个 C 类地址范围。但该网络需要分为 4 个子网：子网 A、子网 B 和子网 C 和子网 D。如果不使用 VLSM，则至少需要将子网掩码设置为 255.255.255.192（11111111 11111111 11111111

11000000）。这样组织的 C 类地址就被划分成了 4 个等大小的子网，每个子网有 61 个可用地址，但实际情况是子网 A 中有 100 台主机和设备，子网 B 有 50 台主机和设备，子网 C 和 D 都有 20 多台主机和设备。显然上述的子网划分方案是不可行的。如果使用 VLSM 则可以很好的解决，因为它允许对网络内的不同子网指定不同的掩码。我们假设该组织的 C 类地址为 192.168.1.0，那么该组织的子网划分方案则如图 3.4 所示。

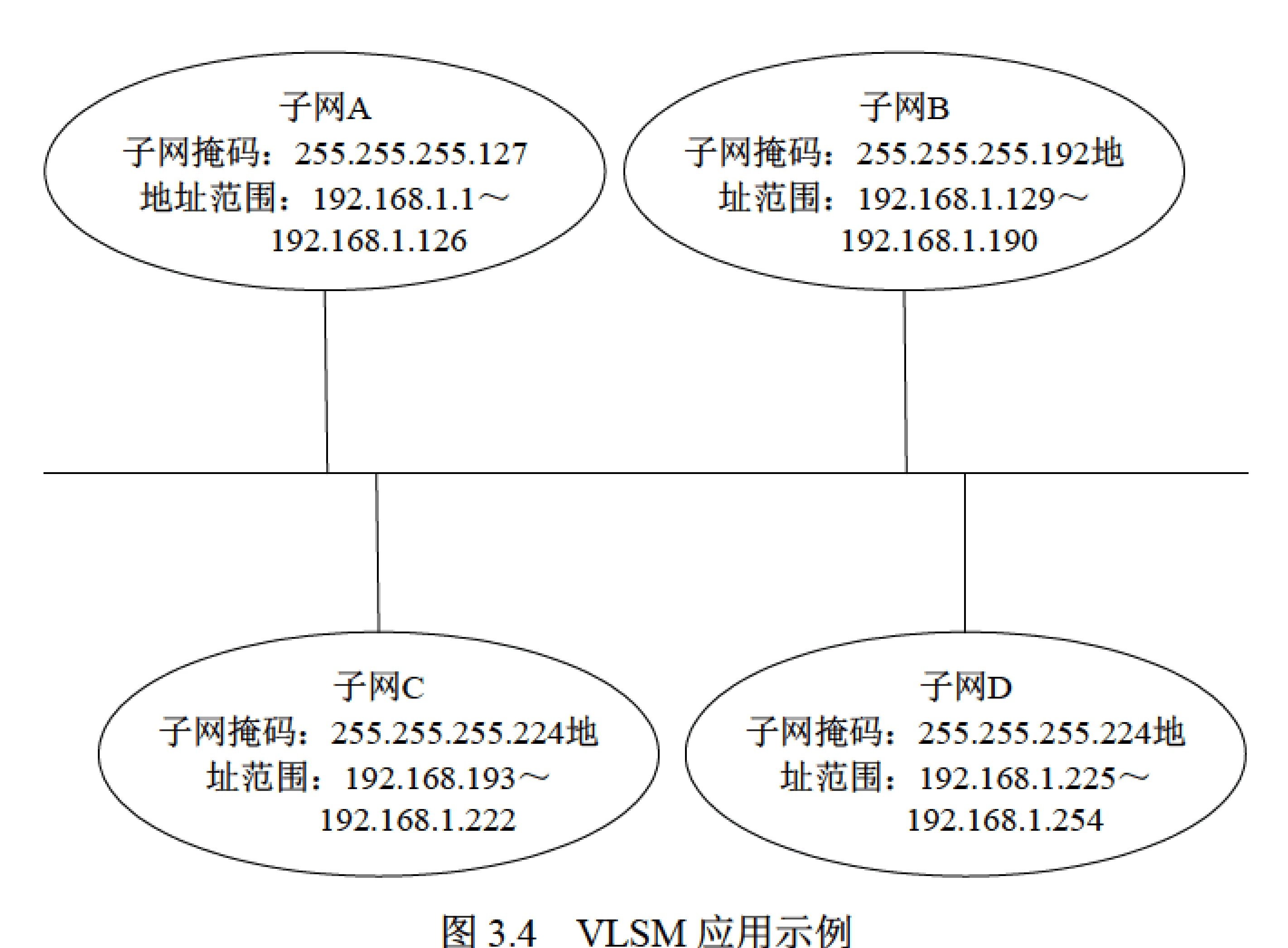


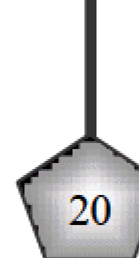
图 3.4　VLSM 应用示例

从图 3.4 可以看出，子网 A 可以容纳 126 个端系统，子网 B 可容纳 62 个端系统，子网 C 和子网 D 可分别容纳 30 个端系统。这种子网划分方案可应用到更大的网络中。

3.2.6　超网技术

虽然使用子网技术可以使 IP 地址得到有效的利用，但还是很难防止 IP 地址资源的耗尽。因为任何一个规模超过 256 的网络都需要使用 B 类地址，所以 B 类地址的消耗速度尤其快，而 C 类地址却得不到应用。解决这个问题的一个办法就是消除 IP 地址中类别的概念。只要是一个连续的地址范围，就可以将其分配给一个适当规模的网络，使地址资源得到充分的利用。例如，我们可以将几个连续的 C 类地址合并在一起，形成一个更大规模的地址范围，这样对于一些网络规模小于该规模的组织就可以使用这个地址范围了。这种技术称之为超网技术。CIDR（无类域间路由）技术实现了超网技术，它不但消除了 IP 地址类别的概念，使 IP 地址得到了更有效的利用，还极大地减小了路由器中路由表的大小，使 IP 数据包的转发变得更加高效。

实际上 CIDR 的概念和子网技术的概念基本相同。在子网技术中，子网的标识由两部分组成：netid 和 subnetid。由于 netid 就是各类 IP 地址中的 netid，而 subnetid 是原 hostid 中的一部分，所以子网技术只能做到把原 IP 地址的网络进行划分而不能将多个网络进行合并。CIDR 将子网划分的概念进一步延伸。它将子网标识的两个部分合并到一起，不加以区分，且对子网标识的长度不作限制。在 CIDR 中将这个子网标识称之为网络部分。例如 192.169.53.27 是一个 C 类地址，如果对其应用子网技术则子网掩码必须是 255.255.x.x，也即



子网标识的长度至少是 25 位。但如果使用 CIDR 技术，则可以将地址表示位 192.169.53.27/20，它标识地址的前 20 位是网络部分，后 12 位是主机部分。那么该地址的网络地址就是 192.169.48.0，而该网络可以容纳 4094 个端系统。

从上述例子可以看出，CIDR 技术可通过设置网络部分的长度来获得各种规模的地址范围，从而使 IP 地址得到了更加有效的利用。

3.2.7 私有网络地址

可能存在这样的情况：某个组织的网络使用了 TCP/IP 技术，但并没有接入到 Internet。如果这样，该组织应该可以使用任何它想用的 IP 地址，只要在组织内部保证 IP 地址不冲突即可。但是，该组织可能会在以后接入到 Internet。因此，如果该组织当前使用的 IP 地址和 Internet 上其他组织使用的 IP 地址冲突的话，日后它接入 Internet 时就必须对组织内所有系统的 IP 地址进行修改并对相关软件的配置做相应修改。为了避免这种情况，IETF 分别从 A、B、C 三类地址中取出一段地址范围保留用作内部网络地址，它们分别是：

10.0.0.0～10.255.255.255

172.16.0.0～172.31.255.255

192.168.0.0～192.168.255.255

这些地址范围是专门用来标识内部网络的，不能用来访问 Internet，因为 Internet 上的路由器是不会转发目标地址在上述三个范围内的数据包的。

3.3 IP 数据包格式

本节将介绍 IP 数据包的具体格式。

3.3.1 网络字节序和主机字节序

在介绍 IP 数据包的格式之前，必须说明整数在计算机中的表示方式的差别及 TCP/IP 协议怎样屏蔽这些差异。

在计算机中，最基本的数据长度单位是字节，8 位。整形数根据其能表示整数范围的不同，有 32 位整数、16 整数和 8 位整数。8 位整型数因为能在一个字节中表示，所以所有的计算机的表示都是一样的。但 16 位整数和 32 位整数必须使用多个字节进行表示，而不同的计算机中对表示整数中的多个字节的解释是不同的。有的将低内存地址的字节解释为整数中低位的字节，这种字节序称为 little indian 序；有的则将高内存地址的字节解释为整数中低位的字节，这种字节序称为 big indian 序。因此，如果直接将整数的本机表示复制后发送给目的地址可能会使两个系统对同一数据包的理解不同。

因此，TCP/IP 协议必须规定一种统一的字节序，以保证各种不同的计算机能对数据包有相同的理解。TCP/IP 协议规定了一种网络标准字节序来表示协议中各种数据的整型数。这样主机或路由器在收到数据包时应现将其中的整形数字段的网络序转换为本机的主机序后再对数据包进行处理；同样，主机和路由器在发送数据包之前应先将数据包中整形数字

段从主机序转换为网络序。网络字节序规定整数中重要的位必须先发送，也即使用 big indian 序。

3.3.2　IP 数据包

IP 协议工作的对象是数据包。和所有的基于数据包的通信协议一样，IP 数据包也由包头和数据组成。图 3.5 是 IP 数据包的具体格式。

<table>
<tr><td>0</td><td>4</td><td>8</td><td>16　19</td><td>24　31</td></tr>
<tr><td>版本</td><td>头部长</td><td>服务类型</td><td colspan="2">总长</td></tr>
<tr><td colspan="3">标识</td><td>标志</td><td>片偏移</td></tr>
<tr><td colspan="2">Time To Live</td><td>协议</td><td colspan="2">头部校验和</td></tr>
<tr><td colspan="5">源IP地址</td></tr>
<tr><td colspan="5">目的IP地址</td></tr>
<tr><td colspan="5">IP选项（如果有）</td></tr>
<tr><td colspan="5">数据</td></tr>
<tr><td colspan="5">...</td></tr>
</table>

图 3.5　IP 数据包格式

数据包的头 4 位是数据包格式的版本号，当前的 IP 版本号为 4。

头部长字段的长度也是 4 位，它标识包头的长度。这里的度量单位是 32 位。从图 3.5 中可以看出，IP 包头中除了 IP 选项字段外，其他的字段都是固定的。常用的 IP 包头（不包含 IP 选项字段）的长度是 20 个 8 位字。因此头部长字段应该为 5。

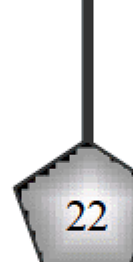
总长字段给出了整个 IP 数据包的长度（以 8 位字计），该长度包括包头和数据包数据。由于总长字段只有 16 位，因此 IP 数据包的总长度最大为 $2^{16}-1$（65 535）个 8 位字。

标识、标志和片偏移字段将在 IP 数据包的分片和重组中说明。

TTL（Time To Live）字段用来防止 IP 数据包永远在 Internet 上。网络上路由器可能会由于某种原因导致路由表出错。这样可能导致 IP 数据包永远在一个环中发送，并最终导致网络中游荡的 IP 数据包越来越多，并最终导致网络崩溃。TTL 可以保证路由器出错时数据包也不会永远在网络中游荡。主机在发送 IP 数据包到网络上时会设置一个 TTL 的值。因为该字段只有 8 位，所以可能的最大值是 255。路由器转发 IP 数据包时至少要将 TTL 的值减 1。如果路由器处理速度很慢，它可以将 TTL 减少该 IP 数据包在该路由器上暂存的秒数。当 TTL 变为 0 时，路由器会将该数据包丢弃并发送一个错误消息给数据包的源地址。这样一个 IP 数据包在经过最多 255 个路由器后会最终消失。这就保证了整个网络不会因某些局部错误和故障而全部瘫痪。

协议字段标识 IP 数据包中的数据来自源地址的上层哪一个协议或应交给目标地址的哪个上层协议处理，如 UDP、TCP 等。

头校验和字段是为了保证 IP 数据包头部的完整性。其计算方法是：计算之前先将头校验和字段设为 0，然后将 IP 头部分为多个 16 位字（网络序）。然后对这些 16 位字进行二进制反码求和。计算结果存在头校验和字段。接收到一个 IP 数据包时，要重新计算头部校验和，



并与收到数据包的头部校验和进行比较。如果两个校验和不相同则标识传输中出现了错误，IP 协议就丢弃该数据包。需要说明的是，IP 协议仅计算 IP 包头的校验和，而不计算数据区的校验和。这样做是为了减少路由器的计算量，同时也让上层协议可以为数据提供自己的校验和计算方法。

IP 选项字段放在后面说明。

3.3.3 服务类型

服务类型字段的长度为 8 位，这 8 位指定了对该 IP 数据包的处理方式。图 3.6 是服务类型字段的组成格式。

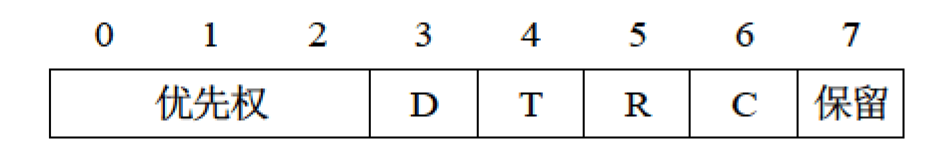


图 3.6 服务类型字段结构

前三位是一个优先权字段，用来指定数据包的重要程度，但现在已被忽略。D、T、R、C 四位分别代表最小延时、最大吞吐量、最高可靠性和最小费用。这四位要么都设为 0，最多设置其中一位为 1。需要注意的是，设置这四个标志位只是让路由器尽可能地按照数据包的性质提供相应的传输质量，这并不保证要求的传输质量能够得到满足，因为 IP 协议只是一种提供最好服务的协议，而这些传输质量的满足还需要依赖硬件条件等许多其他因素。

3.3.4 IP 数据包的分片与重组

IP 包头中有一个总长字段，该字段能表示的最大值为 65 535，因此 IP 数据包的最大长度应该为 65 535 个 8 位字。通常，这个限制不会给上层应用带来问题。但在实际的应用中，却有另外一个长度限制，使得 IP 数据包的长度无法达到这个值。

我们先通过 IP 数据包的实际传输过程来说明这个问题。因为 IP 协议的目的是要向上层屏蔽下层物理网络的差异，它能够在不同的物理层网络上实现。但 IP 数据包的传输最终还是要依赖底层物理网络的传输功能的。这是通过将 IP 数据包作为数据段封装在物理链路层的数据帧中实现的。图 3.7 说明了这种封装。

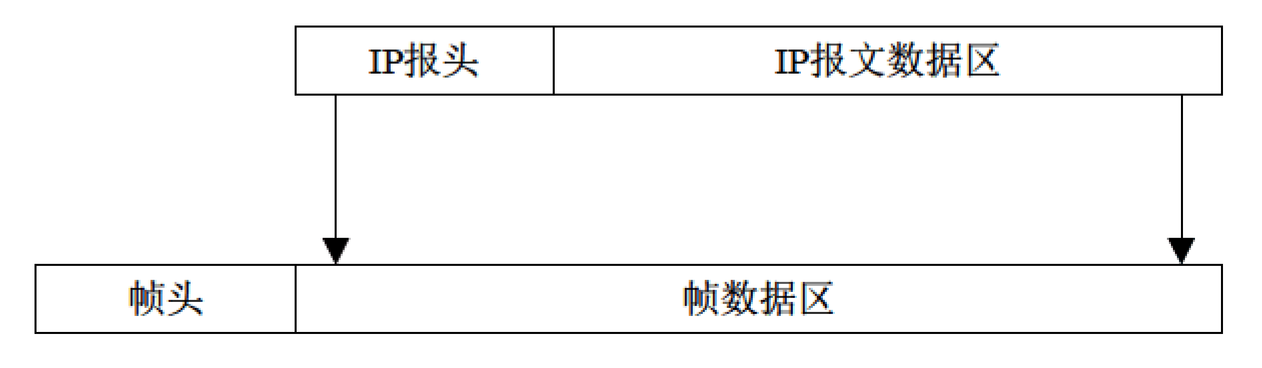


图 3.7 IP 数据包在数据帧中的封装

在很多物理网络中，都对数据帧的长度有限制，而且这个限制大都比 IP 数据包长度的限制小。物理网络的这个限制（对帧数据区）通常都称为最大传输单元（MTU，Maximum Transfer Unit）。例如，以太网的 MTU 为 1500 个 8 位字。因此，如果 IP 数据包的长度大于物理层的 MTU，那这个数据包是不能封装在一个数据帧中的。而由于 IP 协议的目的是为了向上层屏

蔽各物理网络的差异，因此它不能将这个限制强加给上层协议。

为了解决该问题，IP 协议使用了分片与重组的策略。这种策略的原理就是如果一个 IP 数据包无法封装在一个数据帧中，就将数据包分成几个长度小于 MTU 的片，将片封装在帧中进行传输。当这些分解的片都传输到目的地后，再将这些片重新组合成原来的 IP 数据包。分片动作经常会在路由器上发生。因为路由器很可能同时连接在两种不同性质的物理网络上（如以太网和 ATM），这两种网络的 MTU 很可能是不同的。当一个 IP 数据包从 MTU 大的网络发往 MTU 小的网络时，IP 数据包往往就在路由器上进行分片。需要说明的是，IP 数据包的分片可能在 IP 数据包的源主机和网络路由器上发生，但重组只能在目标主机中进行。我们在后面会解释。下面具体介绍这种分片与重组机制。

为了说明这种机制，列举一个例子，并以这个例子为基础来进行说明。图 3.8 是一个网络环境的示意图。

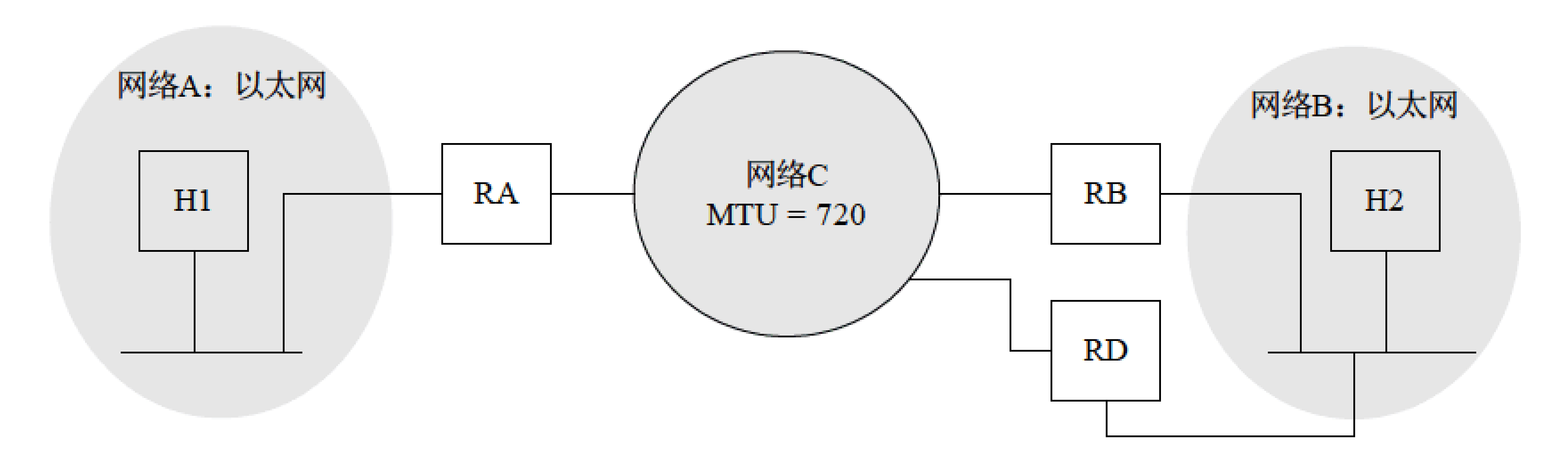


图 3.8　网络环境示例

在图 3.8 中，网络 A 和网络 B 都是以太网，MTU=1500。两个网络之间通过网络 C 相连，网络 C 的 MTU=720。现在假设网络 A 的主机 H1 要发送一个 IP 数据包给网络 B 的主机 H2，这个数据包的数据长 1500B。那么 IP 数据包的总长应该是 1520 个 8 位字，格式如图 3.9 所示。

0	4	8	16	19　24　31
4	5	TOS	1520	
7548			0	0
TTL		协议	头部校验和	
H1				
H2				
数据… len = 1500				

图 3.9　H1 发给 H2 的 IP 数据包

由于这个数据包的长度大于网络 A 的 MTU，所以 H1 的 IP 协议需要将其分片后才能封装在帧中。IP 数据包对数据包进行分片时，每一个分片都会独立地成为一个 IP 数据包。图 3.10 是图 3.9 的数据包分片后的两个数据包 IP1 和 IP2 的格式。

分片后的数据包都有自己的 IP 包头和数据区，因此它们发送到路由器 A 后是作为两个独立的 IP 数据包处理的。从图 3.10 可以看出，分片的数据包头中大部分字段是复制源数据

包头的复制。有几个字段需要注意。首先是数据包总长字段，因为分片后两个数据包的数据长分别是 1480 和 20 个 8 位字，所以数据包 IP1 的长度是 1500，刚好是网络 A 的 MTU，数据包 IP2 的长度是 40。两个数据包的标识字段的值都是 7548。标识字段的值是由数据包的源主机 H1 决定的。数据包标识决定后，对该数据包的分片都会使用该字段的值。这样在目的地址主机上就可以根据标识字段识别哪些 IP 数据包是同一个原 IP 数据包的分片，以便进行重组。在 H1 上通常是通过维护一个全局变量来设置数据包标识，每产生一个原 IP 数据包就将该变量加 1。在 IP 包头中，标志字段由 3 位组成。前两位一般不使用，通常都设为 0。最后一位是结束分片标志，标识该数据包是否是原 IP 数据包的最后一个分片，如果设为 1 则表示该数据包之后还有原数据包的分片，如果设为 0 则标识该数据包是原数据包的最后一个分片。从图 3.10 可以看出，IP1 之后还有分片，而 IP2 是最后一个分片。片偏移字段的值表示该分片的数据相对原 IP 数据包的数据的偏移量，该偏移量从 0 开始计算。因为 IP1 是原 IP 数据包的第一个分片，所以它的片偏移为 0，而 IP2 的数据是原 IP 数据包数据的 1481～1500 的数据，所以它的片偏移是 1480。在数据包的目的地址主机上，可以根据标志字段和片偏移字段判断一个 IP 数据包是否已被分片，是否是原 IP 的最后分片。表 3.2 说明了判断规则。

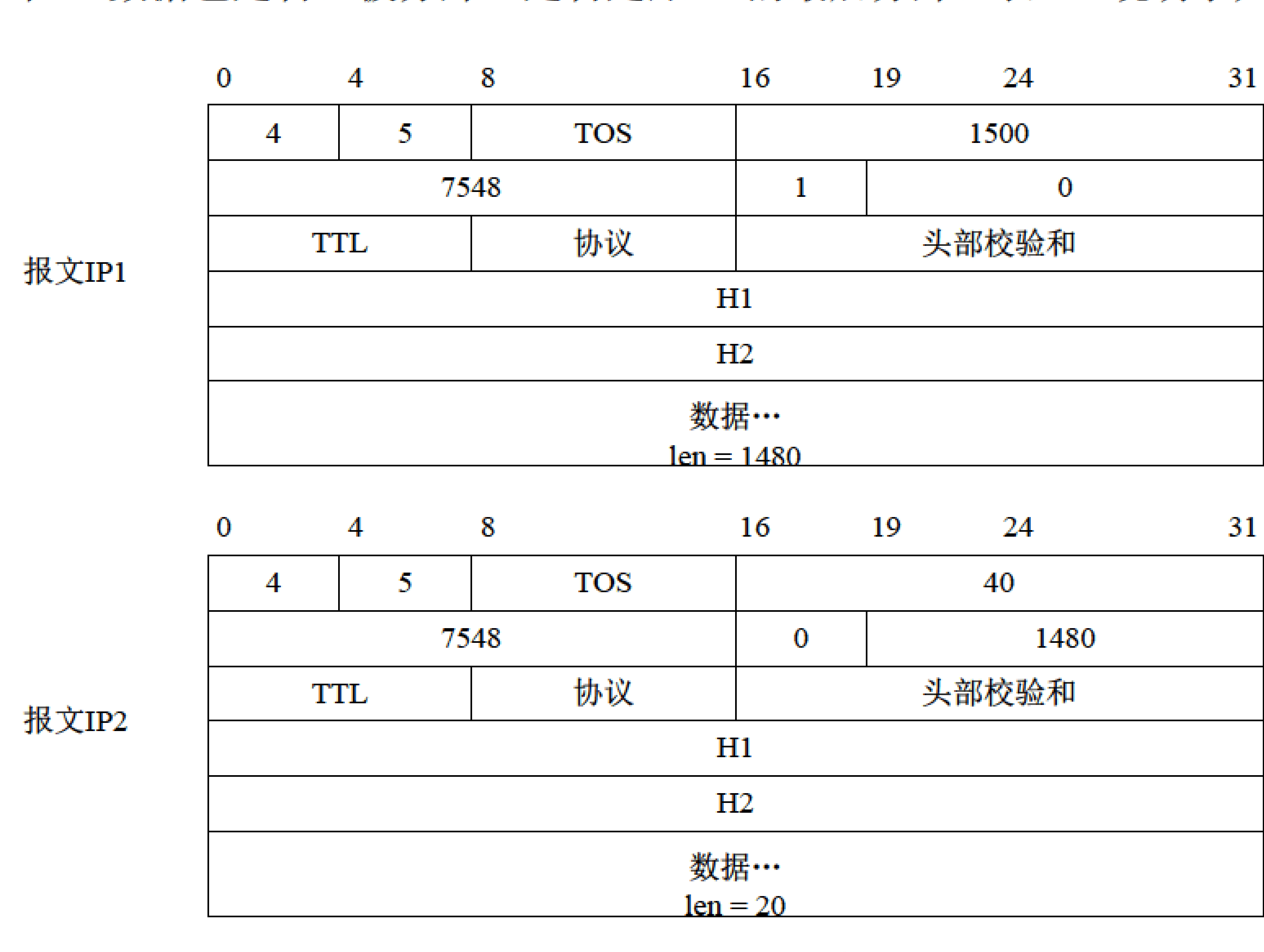


图 3.10　分片后的两个数据包格式

表3.2　目标主机判断IP数据包分片情况方法

标　志　位	片　偏　移	结　　果
最后位为 1	任何	已分片，该数据包不是最后分片
最后位为 0	0	未分片，该数据包是完整数据包
最后位为 0	大于 0	已分片，该数据包是最后分片

通过以上说明可以看出如何在目标主机上对 IP 数据包进行重组。首先，主机收到数据包后先判断该数据包是否已被分片，如未分片则直接交给上层协议处理 IP 数据包中的数据。如

果数据包已分片则根据标识字段将分片进行分组，同标识的分片再根据片偏移重新组合成新的数据。如果收到的片是原 IP 数据包的最后分片，则表示这是原数据包数据的结束。如果一个原 IP 数据包的所有分片在它的第一个分片到达后在一定的时间内都到达，则可以完成 IP 数据包的重组了。如果超过该时间，主机会将该数据包的所有分片都丢弃。因此，IP 数据包分片的机制增加了 IP 数据包传输失败的概率，因为其中任何一个分片传输失败都意味着整个数据包传输的失败。

H1 将 IP1 和 IP2 发送给路由器 RA 之后，RA 还要对 IP 进行分片以适应网络 C 的 MTU，分片结果如图 3.11 所示。

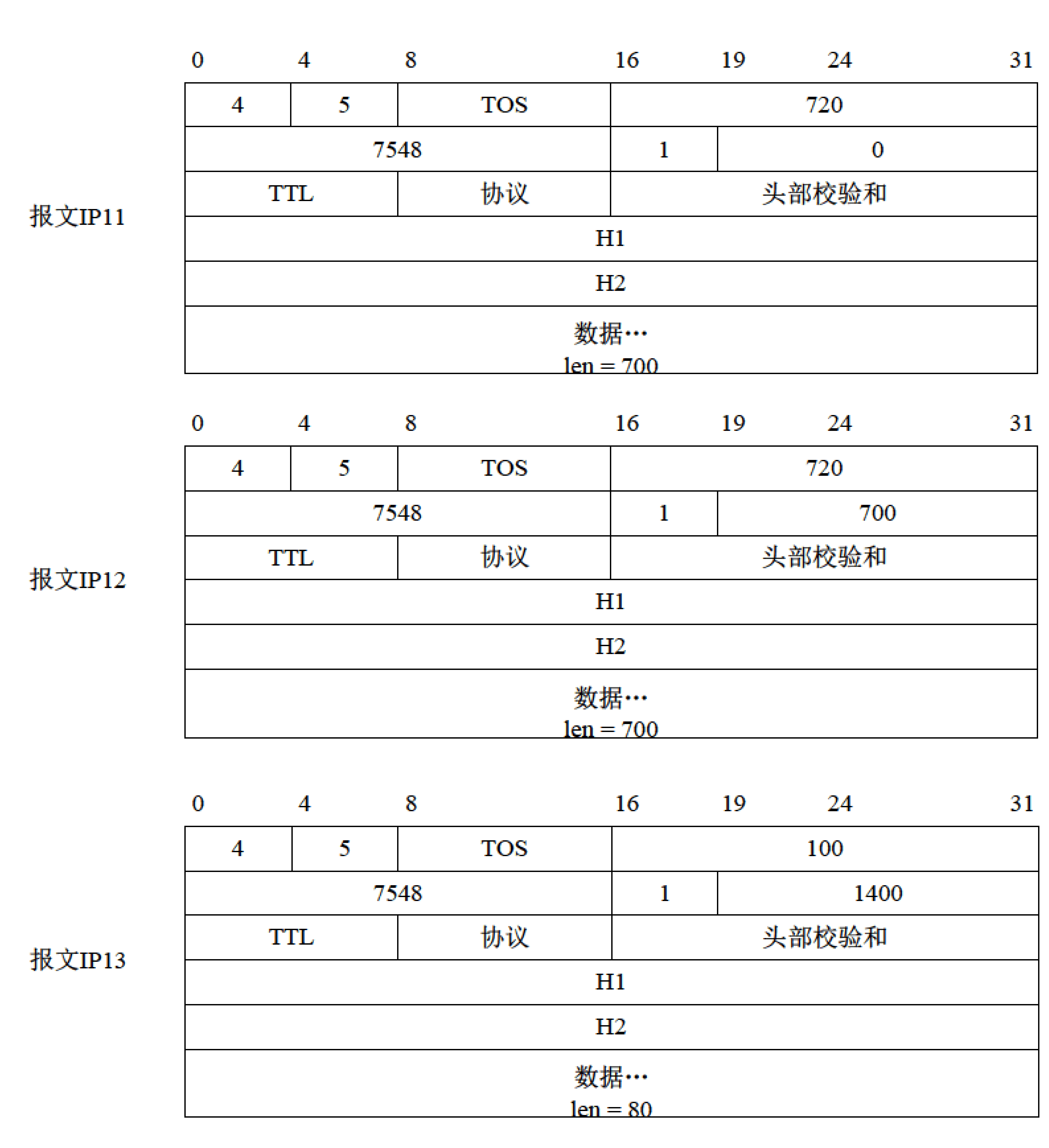


图 3.11　RA 对 IP1 的分片

在对已经是分片的 IP 数据包进行分片时有几点需要注意。首先是标志位的设置，如果 IP 数据包的标志位为 1，那么再分片后的标志位都为 1；如果 IP 数据包的标志位为 0，那么再分片后的分片中最后的分片的标志位为 0，其他的都为 1。其次，对片偏移字段的设置，其值应该是在 IP 数据包的片偏移的基础上的偏移。例如，如果需要对 IP2 进行再分片，那么分片中的第一片的片偏移应该是 1480，而不是 0。

我们再来看为什么 IP 数据包被分片后只能在目标主机上进行重组而不在路由器上进行重组。在路由器上进行重组的优点是可以很好地适应不同物理网络的 MTU。例如，在图 3.8

中路由器 RB 和 RD 在收到 MTU 为 720 的网络 C 的数据包后可以将数据包重新组成更大的数据包，这样多个分片可以通过一个重组后的数据包发送到目标主机 H2，效率更高，且减小了失败的可能性。但我们应该知道路由器处理 IP 数据包的方式。路由器在收到一个 IP 数据包后就会根据数据包的目标地址从路由表中查找应该将数据包发往何处。因为路由器是网络的核心，它处理的数据包数量是非常大的。而重组 IP 数据包时需要根据源地址和数据包标识将数据包的分片进行暂时保存。如果通过路由器的主机数量很多，主机的通信量很大，那么这些分片可能很快就占有了路由器的全部存储空间。

而让所有的数据包都在目标主机进行重组虽然可能会导致最后会有很多的小的分片在 MTU 很大的网络传输，浪费了网络带宽，但这样做有几个很大的好处。首先，这样做减轻了路由器的负担；其次，这样做可以让不同的分片通过不同的路由到达目的地址。上例中的分片可能是通过如图 3.12 的方式到达 H2 的。

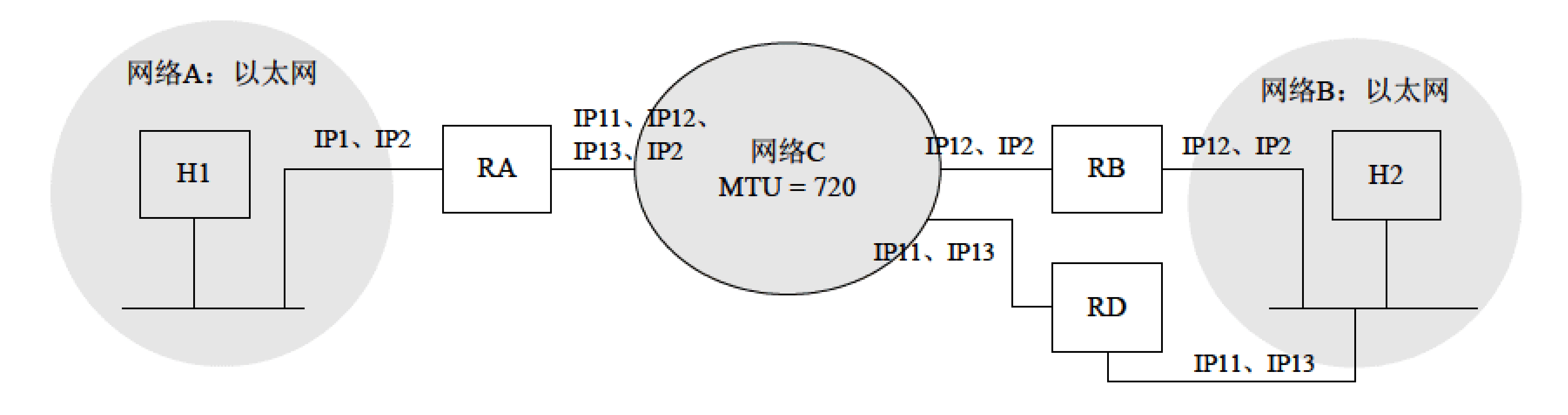


图 3.12　分片的路由

这样既有利于路由器之间的负载平衡，也有利于提高网络的可靠性，因为在有冗余的网络中，分片可以在某条链路故障时从其他链路到达目标地址。

3.3.5　IP 选项

IP 选项字段是可选字段。该字段主要是用来进行网络调试用的，但因为它是整个 IP 协议的一部分，所以所有的 IP 协议的实现中都必须实现对 IP 选项的处理。

IP 选项字段的长度是不定的，根据使用的选项不同而不同。有的选项只含有一个 8 位字长的选项码，有的则除了选项码外还含有变长的选项数据。因为 IP 选项字段的长度是不定的，它的长度很可能不是 32 的整数倍。但应该记得头部长字段的值是以 32 位为单位的。所以，如果 IP 选项字段的长度不是 32 的整数倍，就必须在其后补充若干位让其与选项字段的长度总和为 32 的整数倍。这些补充位须用 0 填充。

不管是什么选项，都以一个 8 位字长的选项码开始。选项码的组成如图 3.13 所示。

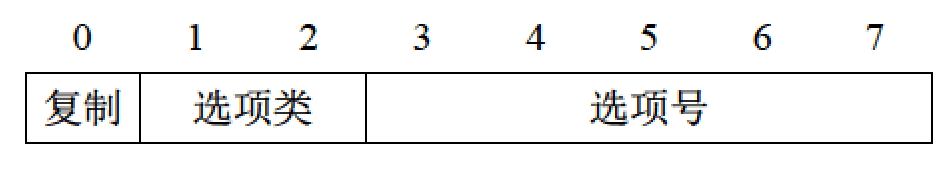


图 3.13　选项码的结构

选项码由三部分组成：复制标志位、选项类和选项号。其中，复制标志位是用来指示对 IP 数据包进行分片时对该选项的处理方式。将该标志位设为 1 表示将 IP 数据包分片时应该将该选项复制到各个分片中；将该标志位设位 0 表示将 IP 数据包分片时只需将该选项复制到

第一个分片中。选项类用来表示该选项所属的类别。总共有四种类别，其中类别 1 和 3 被保留。类别 0 是数据包和网络控制选项，类别 2 是调试和测量选项。选项号用来区别不同的选项。现已定义的各选项中大部分是属于类别 0 的选项，即是用来进行数据包和网络控制的。下面介绍几种常用的选项。

3.3.5.1　记录路由选项

记录路由选项可以用来记录 IP 数据包从源地址到目的地址所经过的路由器。记录路由选项的结构如图 3.14 所示。

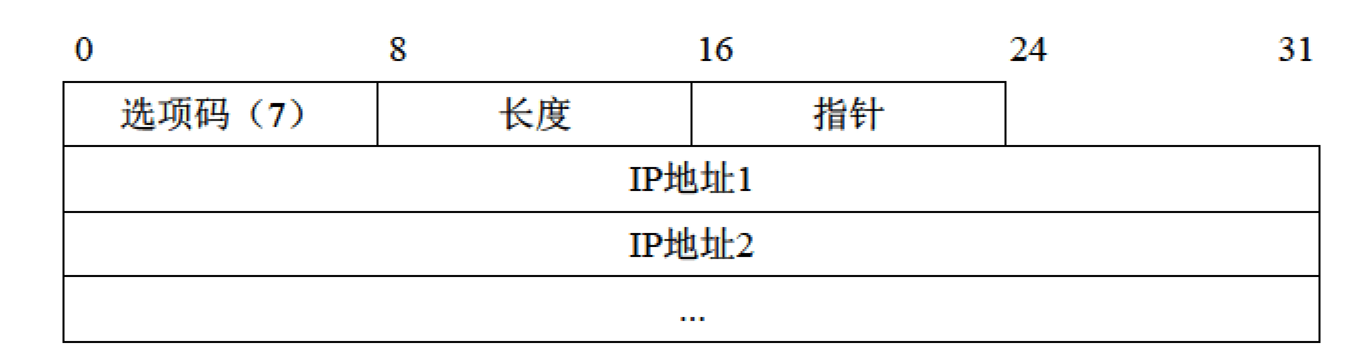


图 3.14　记录路由选项结构

使用记录路由选项的 IP 数据包的处理过程如下：首先源地址主机决定需要记录的路由器（IP 地址）的个数，并分配好空间。将长度字段设置为 IP 地址的个数乘以 4 并加上 3，指针设置为 4。路由器转发数据包时，如果数据包包含了记录路由选项，路由器就将指针和长度进行比较以检查是否选项的 IP 地址表已满。如果 IP 地址表已满，路由器就简单地将数据包发送出去；如果长度大于指针，路由器就将自己的 IP 地址填入到指针所指的位置并将指针值加 4。数据包到达目标主机后，目标主机就可以从 IP 地址表中将 IP 地址去除进行处理了。

但需要注意的是，通常目标主机会忽略记录路由选项。要使用记录路由选项需要源地址主机和目标主机进行配合。源地址主机发送 IP 数据包时需要设置记录路由选项，而目标主机需要对记录路由选项进行处理。还需要注意的一点是，记录路由选项的长度总无法是 32 位的整数倍。

3.3.5.2　源路由选项

源路由选项的作用是用来调试网络，它让源地址主机有能力确定 IP 数据包转发的路径，以确定该路径中的网络是否工作正常。源路由选项的格式和记录路由选项相同，只是选项码不同。源路由选项分为两种：严格源路由（strict source route）和松散源路由（loose source route）。

严格源路由选项的选项码为 137。从该选项码可以看出它的复制标志位为 1，也即该选项需要复制到数据包的所有分片中。带严格路由选项的 IP 数据包的处理过程为：源主机确定要经过的网络的地址列表。将长度设置为 IP 地址的个数乘以 4 并加上 3，并将指针设置为 4。严格源路由选项要求数据包严格按照 IP 地址列表进行转发，即先经过 IP 地址 1，再经过 IP 地址 2，以此类推，且中间不得经过任何其他网络。路由器在转发带严格源路由选项的 IP 数据包时，会将自己的网络 IP 地址与选项中指针所指的 IP 地址对比，如果相同就将指针加 4，表示该网络已经通过。如果路由器找不到符合 IP 地址的网络将该数据包转发，它就会发送一个错误消息给该数据包的源主机。数据包到达目标主机时，指针的值应该大于长度的值。

松散源路由选项的选项码为 131。它和严格源路由选项的功能相似，所不同的是它只要

求数据包能按顺序经过 IP 地址表所指的网络，而不要求两个 IP 地址将不经过其他网络。

3.3.5.3 时间戳选项

时间戳选项的功能与记录路由选项的功能类似，但它还能记录数据包经过相应的路由器的时间。时间戳选项的结构如图 3.15 所示。

0	8	16	24	31
选项码（68）	长度	指针	溢出	标志
IP地址1				
时间戳1				
…				

图 3.15　时间戳选项的结构

长度和指针的作用与记录路由选项的相同。溢出位表示应为选项空间已满而无法进行记录的路由器的数量。标志位则规定了路由器对选项处理方式的不同。下面根据标志位的不同说明带时间戳选项的 IP 数据包的处理过程。

标志位为 0 表示在选项中只记录时间戳而不记录 IP 地址。如果源主机将时间戳选项的标志位设为 0，路由器在转发数据包时如果检查指针小于长度，就将当前的日期和时间记录在指针所指的位置并将指针加 4；如果列表已满，即指针大于长度，则将溢出值加 1。

标志位为 1 表示既记录 IP 地址又记录时间戳。和标志位为 0 的处理不同的是，如果列表还有空间，就将自己的 IP 地址和时间戳一起写入指针所指的位置，并将指针加 8。

标志为 3 表示要记录的 IP 地址已经由源主机确定好了。这时，指针和溢出字段的值就无效了。路由器转发数据包时，会在 IP 地址列表中查找自己的 IP 地址，如果找到就将当前日期和时间记录在该 IP 地址之后；否则就简单将数据包进行转发。

第 4 章　ARP 和 RARP

在第 3 章我们讲过，要将一个 IP 数据包传送到目的地，必须要有一种标识目的地的机制。这种机制就是 IP 地址。IP 地址是一种结构化的二进制地址，IP 协议能够通过该地址找到与该地址相关联的主机或网络设备。在第 3 章中也讲过 IP 数据包的实际传输是通过将其封装在物理网络的数据帧中并让底层物理网络来传输该帧来实现的。物理网络也有一个地址机制，称为物理地址。物理网络就是通过这种物理地址来进行数据帧的发送和接收的。这时就出现了一个问题，因为物理网络是无法识别 IP 地址的，当 IP 协议将数据包交给物理网络发送时，物理网络如何知道该数据包应该发给谁呢。ARP 协议和 RARP 协议就是用来解决 IP 地址和物理地址间的映射问题的。在本章中，先简单介绍物理网络的数据帧传输机制，并具体说明 IP 地址和物理地址的映射问题。然后介绍 ARP 协议，讨论它如何将一个 IP 地址映射到一个与它相关联的物理地址。最后介绍 RARP 协议，该协议的作用与 ARP 协议正好相反，它是用来为一个物理地址关联一个 IP 地址的。

4.1　IP 地址和物理地址映射问题

在本节中，我们以以太网为例，说明物理网络发送和接收数据帧的原理及 IP 地址到物理地址的映射问题。还将讨论对这个问题的两种可能的解决办法。

4.1.1　以太网的传输机制

以太网是一种广播网络，即连接在同一个以太网中的任何主机都能接收到网络上发送的所有数据帧。但主机会检查数据帧中的目的地址，如果该数据帧不是发给自己的，就会将其丢弃。因此连接到以太网的每一个连接口（通常是以太网卡）都会有一个自己惟一的以太网地址。图 4.1 是一个以太网环境示意图。

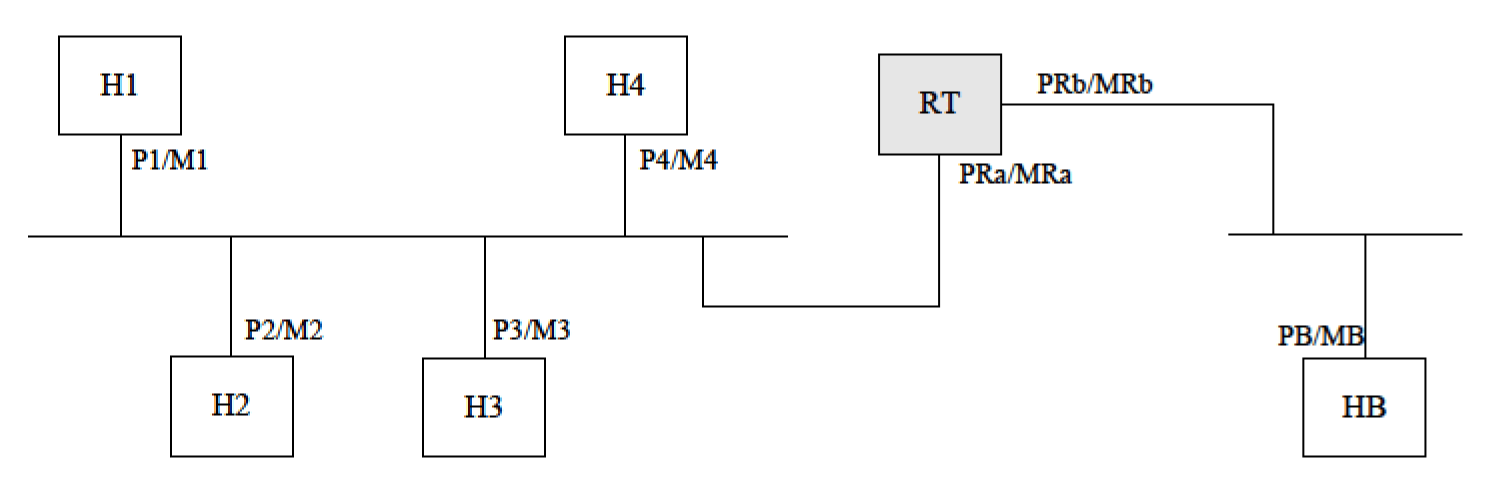


图 4.1　网络环境示例

在图 4.1 中有两个以太网，它们由路由器 RT 相连，且两个以太网上的主机和设备都支持 TCP/IP 协议，所以两个网络之间的主机或内部主机之间都能通过 IP 协议进行通信。每个

连接到网络的接口都有一对地址：IP 地址（Pi）和物理地址（Mi）。以太网的物理地址也称为 MAC 地址。

首先了解以太网网卡的 MAC 地址机制。每个以太网卡出厂时都会有一个固定的全球惟一的 MAC 地址，这样就保证了在任何一个以太网内都不会有重复 MAC 地址的网卡出现。由于以太网是一种广播网，所以只要知道目的主机网卡的 MAC 地址且该网卡已连接到本地主机所在的以太网，就可以发送一个目的 MAC 地址使该 MAC 地址的数据帧到以太网上。目的主机的网卡收到该数据帧后通过检查目的 MAC 地址发现该数据帧是发送给自己的，就会将数据帧中的数据区（也即 IP 数据包）交给上层协议（IP 协议）处理。但是以太网数据帧是无法广播到其他以太网的，如在图 4.1 中 H1 发送的数据帧 HB 是无法收到的。

现在来看不同主机通过 IP 协议通信的处理过程。例如，图 4.1 中主机 H1 要发送一个 IP 数据包给同网络的主机 H4。H1 的 IP 协议先构造一个 IP 数据包，将其目的地址设为 P4，然后将该数据包交给 H1 的网卡处理，并指定它将数据包发往 M4，网卡将数据包封装在数据帧的数据区中，并将数据帧的目的地址设置为 M4，然后将数据帧发送到网络上。H4 的网卡接收到数据帧后发现该帧是发给自己的，就将帧的数据区提取出来，并将其交给上层协议处理，这就完成了整个数据的传输过程。如果是不同网络的主机需要通信，例如 H1 需要发送一个 IP 数据包给 HB，其过程是不一样的。首先 H1 的 IP 协议在构造好 IP 数据包后就将其交给 H1 的网卡，并让其将发送到 MRa。路由器 RT 连接到 H1 所在以太网的网卡接收到发送给自己的数据帧后，就将其中的数据交给上层 IP 协议处理。RT 的 IP 协议发现该数据包是发送给主机 HB 的（因为数据包的目的地址是 PB），就将其交给连接到 HB 所在以太网的网卡处理，并让其将数据发送给 MB。HB 的网卡接收到发给自己的数据帧后就将帧中的数据交给上层 IP 协议处理。

从上述的通信过程可以看出，IP 协议将数据包交给网卡处理时必须告诉网卡将数据发给哪个 MAC 地址。也就是说，IP 协议在发送数据包时必须知道通过哪个 MAC 地址才能到达目的 IP（对同一个以太网中的主机就是主机网卡的 MAC 地址，否则就是相应路由器与自己所在以太网相连的网卡的 MAC 地址）。这就是说，IP 协议必须有一张表，表中的每一条记录都说明了发送给某个 IP 地址的数据包交给网卡处理时，应让其发送到一个相应的 MAC 地址。这张表的产生就是 IP 地址到 MAC 地址的映射过程。

4.1.2 地址映射的可选解决办法

要将 IP 地址映射到一个能用来与其通信的 MAC 地址有两种方法：

第一种方法就是人工建立这张映射表，即主机或设备管理员来设定这张表。用这种方法有几个很难克服的缺点。第一，管理员必须知道一个以太网中所有网卡的 MAC 地址和与其相关的 IP 地址，对于其他以太网的 IP 地址就将其设为路由器相关网卡的 MAC 地址；第二，对于新出现的 IP 地址必须手工添加一条记录；第三，由于主机的网卡是可以更换的，那相应地，MAC 地址也就改变了，但 IP 地址可以不变（也不应该改变，因为 IP 的目的就是要屏蔽物理网络）。这就需要一一更改该以太网中所有主机和设备的映射表中与该 MAC 地址相关的记录。

第二种方法就是利用以太网是广播网的特性，让主机和设备之间相互通知自己的 IP 地址和 MAC 地址的对应关系。这就是 ARP 协议所采用的方法。



4.2　ARP 协议原理

ARP 是 Address Resolution Protocol（地址解析协议）的缩写，它是用来查找同一个物理网络中与一个 IP 地址相关联的物理地址的。

4.2.1　ARP 协议的工作原理

ARP 协议使用一种询问/回答机制。我们以图 4.1 中的网络环境为例说明 ARP 协议的工作原理。主机 H1 要发送一个 IP 数据包给主机 H4，但它只知道 H4 的 IP 地址 P4，而不知道它的 MAC 地址。整个 IP 数据包的发送过程如下。

H1 构造好 IP 数据包，但这时还不能将其交给网卡处理，因为它不知道该发往哪个 MAC 地址。这时 H1 先构造一个 ARP 请求数据包，该数据包中包含了 IP 地址 P4，并留下一个空位表示 P4 的 MAC 地址。H1 的 ARP 协议将该 ARP 数据包交给网卡，让它将该 ARP 数据包作为广播帧发送出去。这样 H1 所在以太网中的所有网卡都会收到该数据帧并对其进行处理，因为它是一个广播帧。网络中的所有网卡在收到该广播帧后会将帧中的数据取出交给上层协议（ARP 协议）处理。ARP 协议在收到这个请求数据包后就将自己的 IP 地址与数据包中包含的 IP 进行比较，如果相同就表示对方在询问自己的 MAC 地址。如果发现不是询问自己的 MAC 地址，ARP 协议会简单丢弃该数据包。因此在上例中，只有 H4 会处理这个 ARP 请求数据包。这时，H4 会将自己的 MAC 地址填在 MAC 地址空位上，并将该数据包改为 ARP 响应数据包，然后让网卡将其发送给主机 H1。地址解析过程如图 4.2 所示。

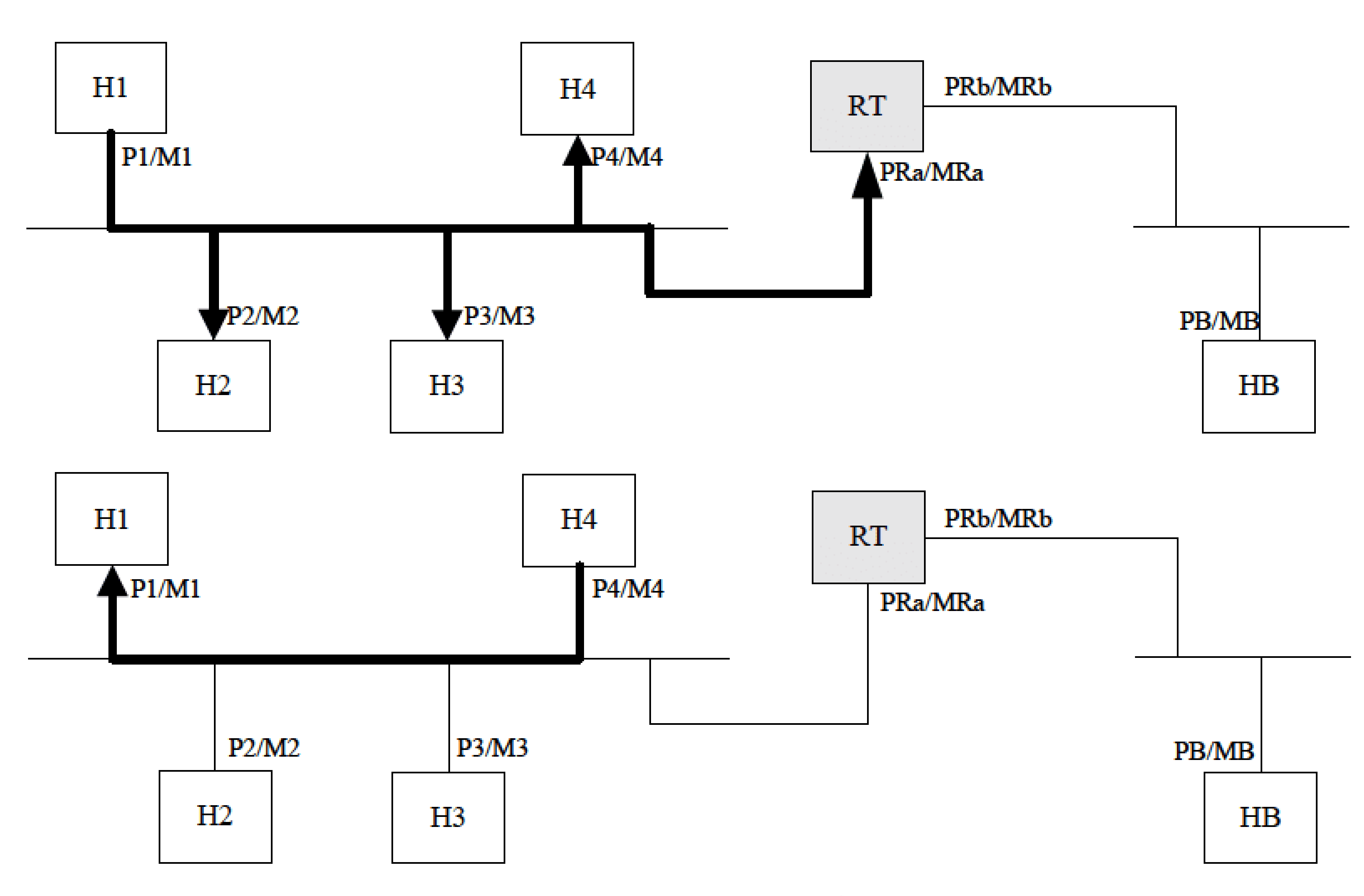


图 4.2　ARP 询问/回答机制

需要说明的是，H1 会在自己发送的 ARP 数据包中包含自己的 IP 地址和 MAC 地址，这样 H4 在向 P1 发送 ARP 响应数据包时就会知道让网卡将其发送给 M1。

对于不在同一个以太网中的通信，该过程略有不同。例如如果 H1 要与 HB 通信。因为 H1 能识别 PB 不是同一个以太网中的 IP 地址（这是因为 IP 地址是一种结构化的地址），H1 不会使用 ARP 协议询问 HB 的 MAC 地址。因为 HB 和 H1 不在同一个网络中，H1 知道必须要经过路由器才能将数据包发送给 HB，因此 H1 会将 IP 数据包发送给路由器 RT。如果 H1 不知道 RT 的 MAC 地址 MRa，它会使用 ARP 协议询问 RT 的 MAC 地址。同样，RT 在转发数据包给 HB 时，如果它不知道 HB 的 MAC 地址 MB，它也会使用 ARP 协议进行询问。从该过程可以看出，ARP 协议只能在同一个物理网络中使用，不属于同一个物理网络的主机永远不会（也不用）知道对方的 MAC 地址。

上面是以一个以太网环境为例说明了 ARP 协议的工作原理。事实上，ARP 可用来将任何的上层协议地址映射到任何种类的物理地址。

4.2.2 减少地址解析需要的通信

如果每次发送一个 IP 数据包都需要进行一次 ARP 请求数据包的广播和 ARP 响应数据包的发送，那么通信发送一个 IP 数据包的代价是非常高的。实际上，可以使用一些措施来减少这种地址解析所需的通信。

如果计算机通过 ARP 协议发现了一个 IP 和 MAC 地址对应关系，它可以记住这种对应关系。在下次还需要往这个 IP 发送 IP 数据包时，就知道需要发往哪个物理地址了，而不用再进行 ARP 询问和回答了。这通常是通过在系统中维护一个 ARP 缓存实现的。在该缓存中，每条记录都有 3 个选项：IP 地址、该 IP 相关联的物理地址和该记录最后更新的时间。之所以需要在 ARP 缓存中维护一个更新时间字段，是因为计算机的 IP 地址或物理地址是有可能会改变的。这种变化可能需要一定的时间，因此每过一定的时间，就应该将一直没有更新的记录从缓存中删除。IP 协议要往某个 IP 地址发送数据包时，先从该缓存查找该 IP 对应的物理地址。只有缓存中没有该 IP 的对应物理地址时才需要使用 ARP 协议进行询问。当 ARP 协议收到一个 ARP 数据包（不管是请求数据包还是响应数据包）时，它可以从中提取出一个 IP 地址和物理地址的对应关系。如果这个对应关系已经在缓存中，ARP 就将记录的最后更新时间更新为当前时间。如果缓存中没有该 IP 的记录，ARP 协议就添加一条该 IP 协议的记录，并将时间字段设置为当前时间。

我们在介绍 ARP 协议的原理时曾说过，如果计算机收到的 ARP 请求数据包不是询问自己的物理地址的，它会将这个 ARP 数据包简单地丢弃。这种说法是不准确的。事实上，计算机会从 ARP 数据包中将数据包的源 IP 地址和源物理地址提取出来，并用它来更新 ARP 缓存，因为这个 ARP 请求数据包说明了这个 IP 地址和物理地址之间的对应关系现在还是正确的。如果随后计算机要与该 IP 地址通信就可以直接从 ARP 缓存中查找到它对应的物理地址了。

另外，计算机在启动时，可以向网络发送一个 ARP 响应广播报，告诉网络中的所有计算机它的 IP 地址和物理地址的对应关系。这样做是因为计算机可能是在更换网卡之后重启的，这时网络中的其他计算机中可能还保存着原来的 IP 地址和物理地址的对应关系。通过启动时的广播 ARP 响应数据包，可以让这些计算机及时将对应关系更正。

4.3　ARP 数据包格式

ARP 协议的数据包格式如图 4.3 所示。

硬件类型		协议类型
硬件地址长	协议地址长	操作
发送者物理地址		
发送者协议地址		
目标硬件地址		
目标协议地址		

图 4.3　ARP 数据包格式

ARP 数据包由 9 个字段构成，但它的长度是不定的。因为可以用来实现任何上层协议地址到任何类型的物理地址的映射，而不同的地址类型其长度是不同的，所以 ARP 协议的长度是由其解析的地址的类型决定的。对于从 IP 地址到以太网 MAC 地址的映射，IP 地址的长度是 4，MAC 地址的长度是 6。

硬件类型字段指明了物理地址的类型，对于以太网，该字段为 1。协议类型字段则指明了上层协议地址的类型，对 IP 协议，该字段为 0x0800。操作字段则指明了 ARP 数据包的类型：ARP 请求数据包的类型是 1；ARP 响应数据包的类型是 2；3 和 4 用来表示 RARP 协议请求数据包和响应数据包，因为 RARP 采用了和 ARP 同样的数据包格式。我们在 4.4 节介绍 RARP 协议。

计算机在发送 ARP 请求数据包时，会将自己的物理地址和 IP 地址填写在发送者硬件地址和发送者协议地址字段中。收到 ARP 请求数据包时，会将发送者硬件地址和发送者协议地址提取出来放入 ARP 缓存中。而在对 ARP 数据包进行响应时，会将自己的硬件地址和协议地址放入目标硬件地址和目标协议地址字段中，并将目标硬件地址和目标协议地址与发送者硬件地址和发送者协议地址呼唤，再将操作字段设为 2。

4.4　RARP 协议

RARP 是 Reverse Address Resolution Protocol（反向地址解析协议）的缩写。RARP 协议起到了为物理地址分配对应的 IP 地址的作用。

在有的计算机系统（如无盘工作站）中没有能断电保存数据的能力。这种系统的启动通常是通过 TCP/IP 文件传输协议从远程服务器上获取启动映像文件来启动计算机系统的。但现在出现了一个问题。因为我们知道，使用 TCP/IP 进行通信必须要有一个 IP 地址。还知道网卡的物理地址是固定在网卡上的，而 IP 地址通常是保存在二级存储器中并在系统启动后载入系统的。因为需要远程启动的系统往往没有二级存储系统，因此在启动之前是没有 IP 地址的。这就出现了矛盾：系统需要使用 TCP/IP 协议来启动，但使用 TCP/IP 的前提条件（IP 地

址）必须在启动之后才有。

RARP 协议的出现就解决了这个矛盾。RARP 协议的实现分为客户端和服务器两部分。需要远程启动的系统必须将 RARP 协议的客户端固化在硬件中（如 ROM），而服务器端并不是所有的 TCP/IP 的实现都包含了 RARP 协议的。TCP/IP 通常是作为系统的核心实现的，而 RARP 服务器通常是作为一种服务提供的。

RARP 协议的原理很简单：需要知道自己 IP 地址的计算机发送一个 RARP 请求数据包给 RARP 服务器，服务器向该计算机发送一个 RARP 响应数据包，响应数据包中包含了请求计算机的 IP 地址。请求计算机在获得了自己的 IP 地址之后就可以用它来与文件服务器通信获取自己的启动映像文件了。至于与文件服务器之间的通信机制细节在此不加以讨论。

RARP 数据包的格式与 ARP 数据包的格式相同。现在来详细说明 RARP 协议的工作 过程。

请求计算机构造一个 RARP 请求数据包。在该数据包中，计算机将发送者硬件地址和目标硬件地址都设为自己的物理地址。然后将该数据包广播到网络中。网络中所有的计算机都能接收到该数据包，但只有 RARP 服务器会处理。RARP 服务器将请求计算机的 IP 地址放入数据包的目标协议地址字段中，并将数据包类型改为 4（响应）。然后将响应数据包发送给请求计算机。

在远程启动过程中，RARP 请求的成功是必不可少的。但如果出现网络故障或 RARP 服务器失效时，RARP 请求将会失败。通常请求计算机对这种情况的处理是使用超时重发机制。如果经过多次重发，系统机会向用户报告一个错误。因为在局域网中网络故障发生的可能性比较小，但服务器发生故障或重新启动的可能性是比较大的。所以为了保证服务器失效时计算机能正常启动，通常都在一个网络中放置多个 RARP 服务器。但这样做也有它的不足。因为所有的 RARP 服务器都会对 RARP 请求产生响应，这就会造成网络流量的增多。

第 5 章　ICMP 协议

因为 IP 协议是提供不可靠传输服务的，因此源地址发出的 IP 数据包很可能无法到达目标地址。但发生这种情况的原因是很多的，可能是目标主机根本不存在，也可能是传输途中的某个链路中断等。那么在 IP 数据包无法传送到目标地址时，发送方怎样才能知道是什么原因造成的呢？ICMP 协议就是用来探测并报告 IP 数据包传输中产生的各种错误的。在本章中，我们将介绍 ICMP 协议。

5.1　ICMP 协议的作用与原理

ICMP 是 Internet Control Message Protocol（互联网控制消息协议）的缩写。ICMP 协议就是用来探测并报告 IP 数据包传输中产生的各种错误的。

我们都知道 IP 协议的工作原理。IP 数据包在网络上的传输是通过路由器对数据包的转发来完成的。如果在 IP 数据包的传输过程中，某个路由器因为某种原因无法转发收到的数据包时，数据包就会被丢弃。但这时数据包的发送站无法得知传输出错，更不知道出错的具体原因。而一个有效的错误检查与报告机制对 TCP/IP 协议是非常重要的，因为它可以使我们在网络发生故障时知道故障的具体原因与位置。

ICMP 协议就是这样一种能让我们对网络进行调试的报错机制，它能让发现错误的路由器向数据包的源站发送一个出错消息，报告出错原因。这样源站就可以根据不同的错误采取不同的措施进行处理。需要指出的是，ICMP 的错误报告只能通知出错数据包的源主机，而无法通知从源主机到出错路由器途中的所有路由器。例如：在图 5.1 中主机 H1 向 H2 发送一个 IP 数据包，路由器 RC 发现该数据包无法将该数据包转发到 H2。我们看看通过 RC 接收到的数据包能提取哪些信息。RC 能知道数据包的源地址和目标地址，但它无法知道该数据包到达本路由器时途中经过了哪些其他的路由器。因此 RC 只能将出错消息发送给数据包的源地址 H1。

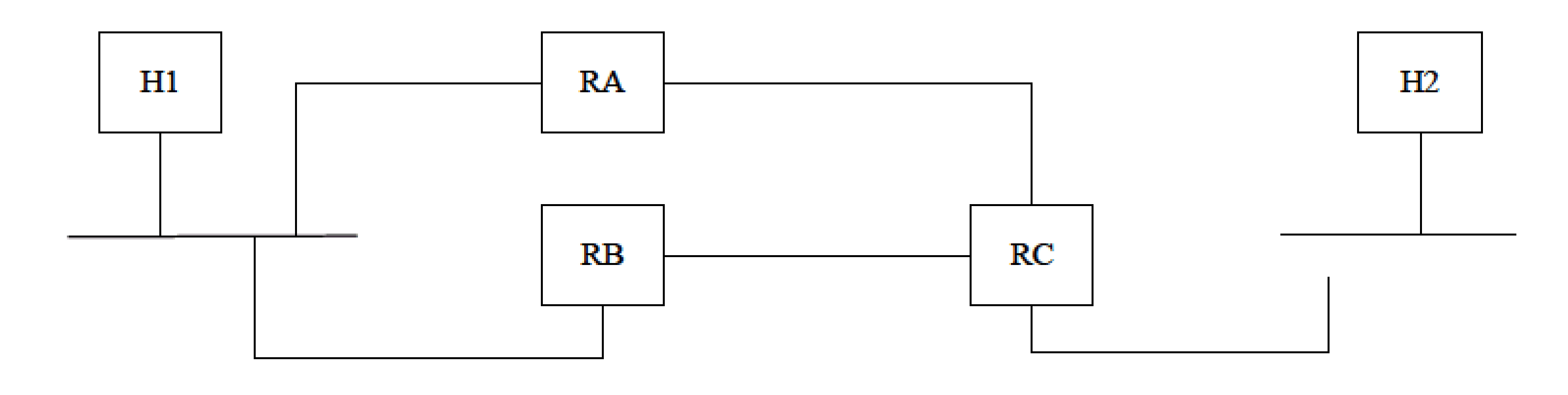


图 5.1　IP 数据包出错示例

从图 5.1 中还可以看出 ICMP 数据包应该如何传输。如果 RC 要向 H1 发送一个 ICMP 数据包，那么该数据包必须通过路由器 RA 或 RB，即要由它们之中的一个将数据包转发给主

机 H1。但路由器只会转发 IP 数据包，所以应该将 ICMP 数据包封装在 IP 数据包中才能将它传输到目的地。事实上 ICMP 数据包确实是封装在 IP 数据包中的，如图 5.2 所示。

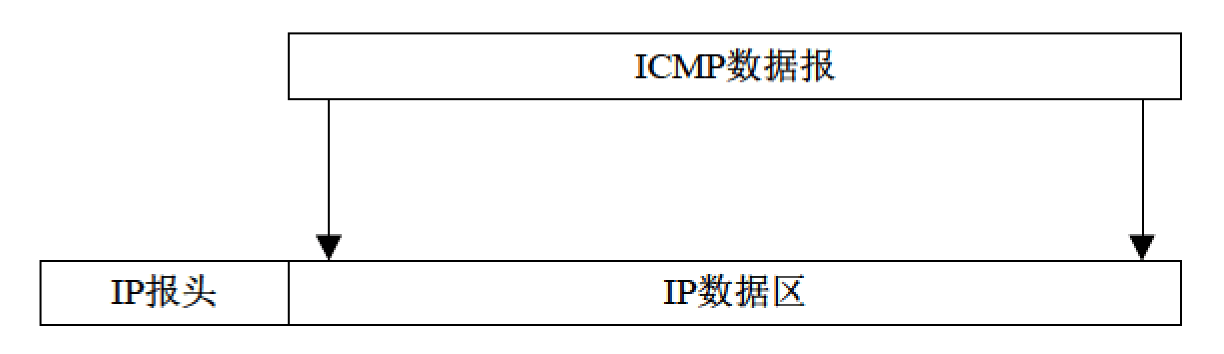


图 5.2　ICMP 数据包的封装

因为 ICMP 数据包是封装在 IP 数据包中的，因此 ICMP 数据包的传输也有可能会出错。这时就需要为这个 ICMP 数据包产生另一个 ICMP 数据包，并将它发送给源 ICMP 数据包的源路由器。在一个已经很繁忙的网络中，IP 数据包的出错率是比较高的，也即发送的 ICMP 数据包比较多，如果再为这些 ICMP 数据包产生新的 ICMP 数据包，就会加重网络负载。使得本已阻塞的网络阻塞得更严重。因此 ICMP 协议规定，如果传输 ICMP 数据包的 IP 数据包出错，不能为该数据包产生新的 ICMP 数据包。

5.2　ICMP 数据包的格式

因为传输错误的种类多种多样，ICMP 协议要报告这些错误就必须根据不同的错误采用不同的格式。但各种 CIMP 数据包都有一个共同的 ICMP 头部。图 5.3 说明了 ICMP 数据包的头部格式。



图 5.3　ICMP 数据包格式

类型字段定义了各种不同的 ICMP 数据包，不同的 ICMP 数据包起到不同的作用，也有不同的格式。表 5.1 是已定义的各种 ICMP 类型。

表5.1　ICMP数据包类型

类型字段值	描　　述
0	回显（echo）应答
3	目标不可达
4	源端关闭
5	重定向
8	回显请求
9	路由器通告
10	路由器请求
11	超时

续表

类型字段值	描　　述
12	数据包参数错误
13	时间戳请求
14	时间戳应答
15	信息请求（作废）
16	信息应答（作废）
17	地址掩码请求
18	地址掩码应答

虽然大多的 ICMP 数据包都是用来报错的，但有的是用作探测和获取信息的。代码字段是用来说明错误的具体原因的，将在 5.3 节具体介绍各种 ICMP 数据包时详细说明。

5.3　各种 ICMP 数据包

由于 ICMP 数据包的种类较多，在本节将按照各种 ICMP 数据包的不同功能分小节对其中常见的几种进行详细说明。

5.3.1　回显请求与应答

回显请求和应答数据包用来测试从发送主机到接收主机的网络链路是否完好以及目标主机的 TCP/TP 协议是否工作正常。图 5.4 是回显请求与应答数据包的格式。

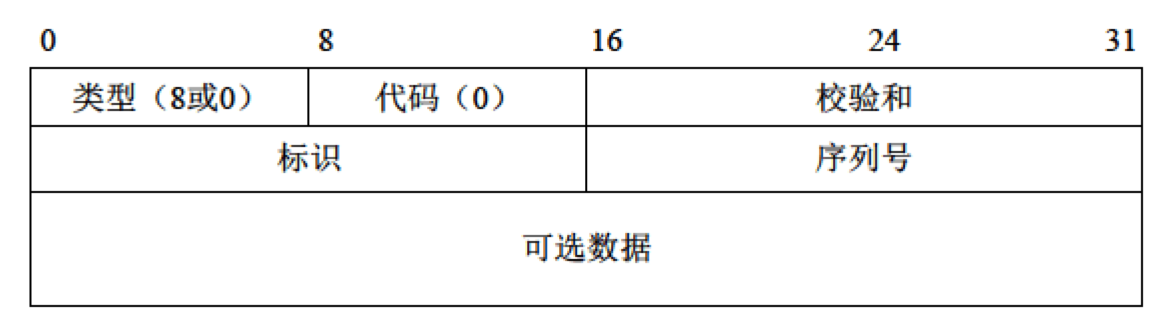


图 5.4　ICMP 回显请求与应答数据包格式

回显请求与应答数据包的代码字段都必须设为 0。可选数据字段是一个变长的字段，发送方可以用任何数据填充该字段，当然，该字段的长度也可以为 0。标识和序列号字段是发送方用来匹配请求数据包和应答数据包的。回显请求与应答的工作过程如下：发送方构造一个回显请求数据包，它可以设置任意长度的可选数据字段，其中可以随意填写数据。目标主机收到该数据包后将类型字段改为 0 再发送给请求数据包的发送方。一旦请求发送方接收到请求数据包的应答数据包，它能肯定两个问题：首先，从发送方到接收方的网络通路工作正常；第二，目标主机的 TCP/IP 协议工作正常。

5.3.2　目标不可达错误

当一个路由器无法将一个 IP 数据包转发到目的地时，它会将该数据包丢弃，但同时它会向该数据包的源主机发送一个目标不可达的 ICMP 数据包。该数据包起到了报错的作用。

ICMP 目标不可达数据包的格式如图 5.5 所示。

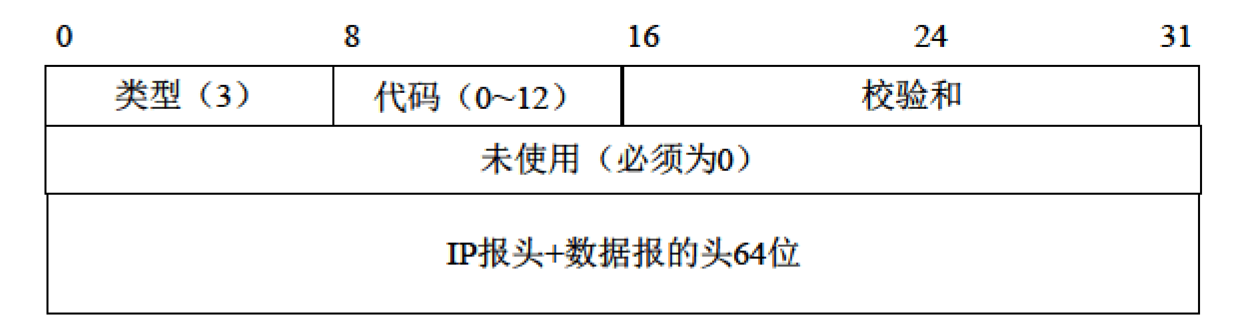


图 5.5 ICMP 目标不可达数据包格式

类型字段为 3，标识这是一个 ICMP 目标不可达数据包。代码字段的值进一步说明了出错的具体原因，其取值范围为 0～12。各值的含义如表 5.2 所示。

表5.2 ICMP目标不可达数据包代码字段值

代 码 值	描 述
0	网络不可达
1	主机不可达
2	协议不可达
3	端口不可达
4	需要分片，但设置了 DF（不分片）位
5	源路由失败
6	未知的目标网络
7	未知的目标主机
8	源主机孤立
9	禁止与目标网络通信
10	禁止与目标主机通信
11	指定服务类型的网络不可达
12	指定服务类型的主机不可达

另外我们注意到 ICMP 目标不可达数据包中最后一个字段，该字段包含了出错 IP 数据包的 IP 头部和 IP 数据包数据部分的头 64 位。之所以要包含这个字段是为了能让 IP 数据包的发送者知道是哪个数据包出错。

5.3.3 源端关闭

路由器转发 IP 数据包时是仅仅根据单个数据包的信息进行转发的，因此它无法为某个数据包预留一定的资源。由于路由器的资源（内存，计算时间等）是有限的，所以在网络数据流量比较大时，路由器为了转发数据包可能会耗尽其所有资源，这种情况称为阻塞。出现阻塞就是路由器能力有限，无法转发所有需要转发的数据包。这时路由器会选择丢弃一些数据包。但仅仅这样做还是无法减轻网络阻塞的情况，因为发送端在发出数据包后如果无法收到响应往往会发送更多的数据包到网络上，使得网络更加拥挤，使网络的阻塞更加严重。因此路由器在丢弃数据包时会发送一个信息到数据包的发送方，通知它网络已阻塞，请缓发数据包。用来发送这个通知的就是 ICMP 源端关闭数据包。该数据包格式如图 5.6 所示。

0	8	16	24	31
类型（4）	代码（0）	校验和		
未使用（必须为0）				
IP报头+数据报的头64位				

图 5.6　ICMP 源端关闭数据包格式

从图中可以看出，ICMP 源端关闭数据包的格式和目标不可达数据包格式相同，只是类型和代码字段的值不同。ICMP 源端关闭数据包也在最后一个字段包含了被丢弃的 IP 数据包的 IP 头部和数据部分的头 64 位。

5.3.4　超时错误

在 IP 数据包中有一个 TTL 字段。IP 数据包每被路由器转发一次该字段的值就至少减 1。当该字段的值减少到 0 时路由器会将该数据包丢弃。但这种情况是很少发生的，因为 IP 数据包都会将该字段的值设置的足够大，使得该数据包被转发到目标主机时该字段的值仍然大于 0。但有时也会发生这种情况，例如几个路由器组成一个环，而且路由器中的路由表有错误，这时在其中转发的数据包可能一直在这个环中直到 TTL 字段减为 0。在将 TTL 为 0 的数据包丢弃之前，路由器需要通知数据包的源主机数据包转发超时了。

由于一个 IP 数据包在转发到目标主机的过程中可能被分成了多个片，而这些分片是单独路由的。因此很有可能一个 IP 数据包的多个分片只有部分能到达目标主机而其他的可能丢失了。目标主机在对一个 IP 数据包的多个分片进行重组时，如果等待一定的时间后剩下的分片还没有到达，目标主机就认为这些分片不会到达，并将已收到的分片丢弃。在丢弃这些分片之前，目标主机会通知源主机 IP 数据包的分片在进行重组时超时了。

不管是上述两种情况的哪一种，路由器和目标主机都是通过向源主机发送一个 ICMP 超时数据包来通知它的。该数据包的格式如图 5.7 所示。

0	8	16	24	31
类型（11）	代码（0或1）	校验和		
未使用（必须为0）				
IP报头+数据报的头64位				

图 5.7　ICMP 超时数据包格式

其中代码字段说明了超时的具体原因。0 标识 TTL 值减为 0，而 1 表示分片重组超时了。

5.3.5　数据包参数问题

如果路由器或主机发现一个收到的 IP 数据包的格式不符合要求，它就会向源主机发送一个报错，并指出数据包中的什么字段格式或值不正确。这就是 ICMP 参数错误数据包的作用，该数据包的格式如图 5.8 所示。

在 ICMP 参数错误数据包中，代码字段的值表示了 IP 数据包格式错误的类型。0 表示 IP

数据包中头部某个字段的值设置错误，这时指针字段的值就表示是 IP 头部的那个八位字的值出错了。代码 1 表示 IP 头部需要设置某个选项但没有设置，这时指针字段没有意义。

0	8	16	24	31
类型（12）	代码（0或1）	校验和		
指针	未使用（必须为0）			
IP报头+数据报的头64位				

图 5.8 ICMP 参数错误数据包

5.3.6 获取子网掩码

关于子网掩码的作用与原理在第 3 章已经详细讲述过。现在的问题是一个使用了子网掩码技术网络中的主机要与该网络中的其他主机或者与该网络外面的主机通信必须知道该网络的网络掩码。在常见的系统中，这个掩码是可以认为设置的，但这就要求系统管理人员知道该网络的掩码。ICMP 可以自动获取网络使用的子网掩码，这是通过发送子网掩码请求数据包并接收响应数据包实现的。这两种数据包的格式如图 5.9 所示。

0	8	16	24	31
类型（17或18）	代码（0）	校验和		
标识		序列号		
地址掩码				

图 5.9 ICMP 子网掩码请求与响应数据包

请求主机可以向路由器或者直接广播该数据包，路由器会将网络的子网掩码通知给该主机。

第 6 章　路由协议

我们讲到 IP 协议包的传输时提到当主机要与一个与自己不在同一个网段的主机通信时，它往往是先将数据包发送给自己所在网络中的路由器，然后由不同的路由器将数据包转发到目标主机。可能有的读者会很疑惑，路由器怎么知道目标主机在什么地方，应该将数据包往哪转发呢？本章就来仔细研究这个问题。

通常，在配置主机系统的网络配置时使用的都是静态路由及在配置接口时，以默认方式生成路由表项（对于直接连接的接口），并可以通过命令增加表项（通常从系统自引导程序文件），或是通过 ICMP 重定向生成表项（通常是在默认方式出错的情况下）。在网络很小，且与其他网络只有单个连接点且没有多余路由时（若主路由失败，可以使用备用路由），采用这种方法是可行的。如果上述三种情况不能全部满足，通常使用动态路由。本章要讨论的路由协议就是用于动态路由的，它能在路由器间交换网络拓扑结构和可达性的信息。

在本章中先简单介绍路由器的工作原理，主要目的是看看路由器要正常工作需要哪些条件，这是路由协议的目标，然后具体讲述几种常见的路由协议。

6.1　路由器的工作原理及路由协议

路由器的作用就是在不同的网络之间转发 IP 数据包，使得该 IP 数据包能够正确地、尽量快地到达它的目的地。那么路由器具体是如何进行数据包的转发的呢？为什么需要有路由协议呢？本节将详细说明这些问题的答案。

6.1.1　路由器的工作原理

图 6.1 说明了路由的工作方式和作用。

在图中路由器 Ra 有 4 个端口连接 4 个不同的的网络，路由器 Rb 连接了 2 个不同的网络，而路由器 Ra 和 Rb 之间是通过一个共同的网络相连的。现在有一个 IP 数据包从 Ra 的左侧的端口进入路由器，该数据包的目的地址是 202.119.15.137。因为 Ra 的下面的端口连接到了网络 202.119.15.0/24，所以 Ra 直到主机 202.119.15.137 是直接连接到该路由器的，所以它将该数据包直接转发给了目的地址。又有一个数据包从上面的端口到达路由器 Ra，其目的地址是 202.119.12.24。这时，路由器要将该数据包转发进行正确的转发它就必须知道要通过路由器 Rb 才能到达目的地。我们现在不管 Ra 是如何知道这一点的，假设它已经知道要到达网络 202.119.12.0/24 就必须通过路由器 Rb。这时，Ra 会将该数据包通过右边的端口转发给路由器 Rb。Rb 在收到该数据包后再将它从右边端口转发给目的地址 202.119.12.24。

这就是路由器大致的工作过程。现在来看看路由器实现这种功能的具体机制。

路由器收到一个数据包时，必须要能确定应该从哪个端口将该数据包转发出去。例如图

6.1 中，对于发送到 202.119.12.24 的数据包，Ra 必须将其从右面的端口发送该 Rb，而不能从下面的端口发出。通常，路由器是通过查找一张路由表来确定转发端口的。在路由表中，每一项都说明了通往一个目标地址应该通过哪个端口进行转发，是直接转发给目的地还是需要通过下一个路由器再进行转发。因此，一个基本的路由表的表项包含以下几个字段：目的地址，下一站的地址，转发端口。例如在图 6.1 的例子中，Ra 的路由表中至少需要有表 6.1 所示的几项。

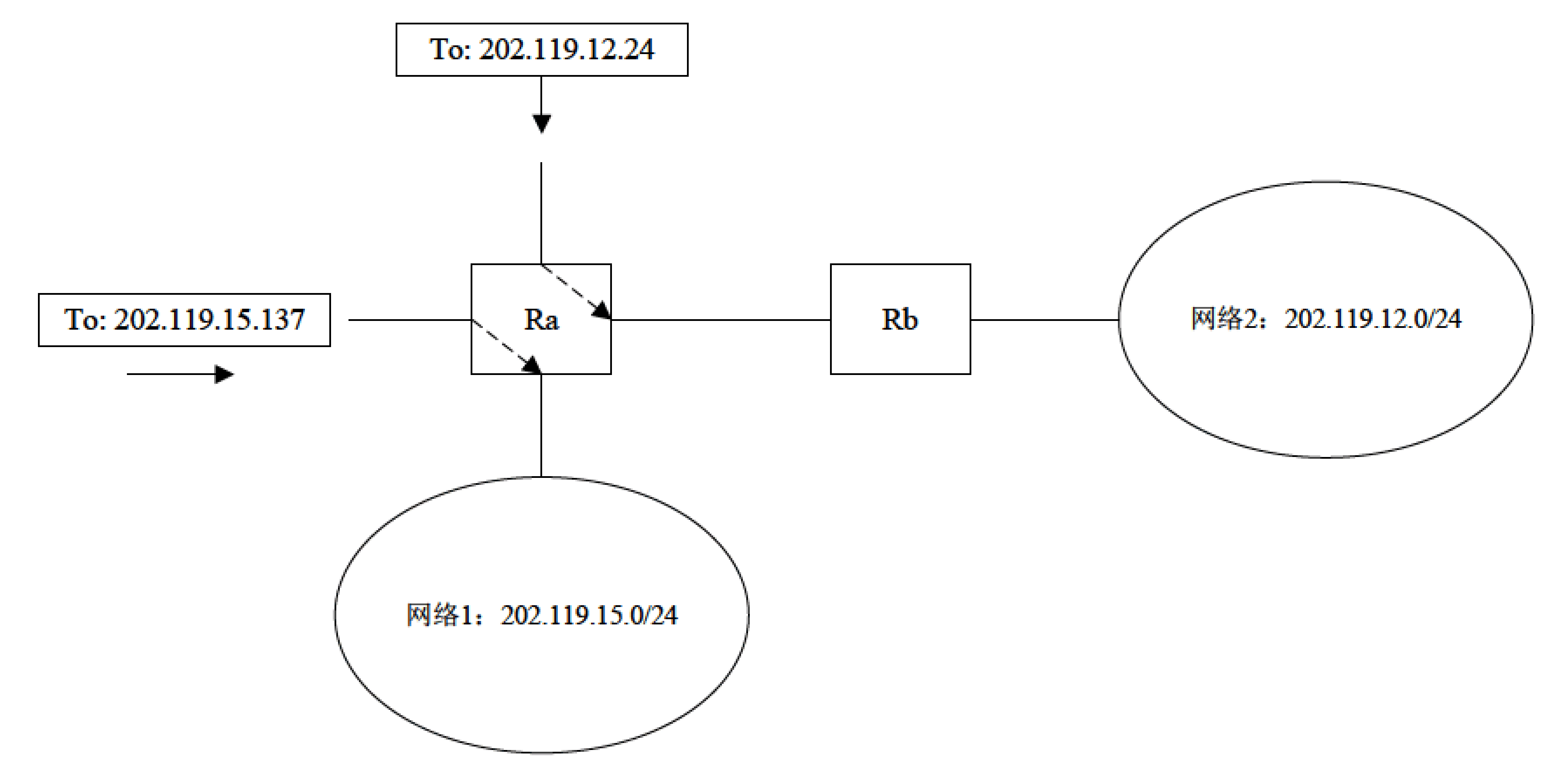


图 6.1　路由器作用示意图

表6.1　路由器Ra的部分路由表

目的地址	转发下一站	端　　口
202.119.15.0/24	直接转发	下
202.119.12.0/24	Rb	右

从表 6.1 可以看出，路由表中的目的地址是网络地址，而不是某个特定的主机地址。因为这样可以极大的减小路由表的大小，提高查找的效率。现在来看看 Rb 对图 6.1 所示的两个数据包的不同的处理过程。

首先，我们看看发往 202.119.25.137 的数据包。Ra 在收到该数据包后通过查找路由表发现这个地址是网络 202.119.15.0/24 中的一个地址，需要通过下方的端口进行转发。注意，在转发下一站字段中 202.119.15.0/24 表项注明的是直接转发。这个字段很重要，因为 IP 数据包是通过底层的物理数据帧进行传输的，所以路由器在将数据包转发出去之前必须确定接收该数据包的下一站的物理地址。因为现在要转发的数据包是直接转发，所以路由器必须将它的物理地址设置为主机 202.119.25.137 的物理地址，这样数据包发送到网络 202.119.15.0/24 中之后，只有 202.119.25.137 会接收到该数据包。

对于发往 202.119.12.24 的数据包，Ra 通过查找路由表发现应该通过右端口将其转发给路由器 Rb。这时 Ra 就将数据包的物理设置为 Ra 的左端口的物理地址并将其从右端口发送到该端口所连接的网络上。在该网络上，只有 Rb 会接收该数据包。Rb 收到数据包之后会进行同样的处理。

上面看到了路由器的内部通过使用路由表进行数据转发的机制，现在有另外一个问题出现了，即路由器的路由表是怎样建立的？这需要两种手段。

第一种手段是路由器根据路由器各端口直接连接的网络自动产生一些路由表项。例如图 6.1 的例子中，路由器 Ra 的下面的端口直接连接到网络 202.119.15.0/24，那么它就可以产生表 6.1 中的第一条表项了。同样，路由器 Rb 的右端口是直接连接到网络 202.119.12.0/24 的，这样 Rb 也可以产生一条路由表项，如表 6.2 所示。

表6.2　路由器Rb自动产生的路由表项

目的地址	转发下一站	端　　口
202.119.12.0/24	直接转发	右

第二种手段就需要路由协议的帮助了。从表 6.1 可以看到，Ra 必须知道通往网络 202.119.12.0/24 必须经过路由器 Rb。那么 Ra 如何知道这一点呢，方法只有一个，就是 Rb 告诉 Ra 这一点。Rb 就是通过路由协议将这条信息告诉 Ra 的。

通过这两种手段，路由器就可以建立一张完整的路由表了。

6.1.2　路由协议的作用及分类

路由协议的作用在于它是路由器能够与其他的路由器交换有关网络拓扑和可达性的信息。任何路由协议的首要目标是保证网络中的所有路由器都具有一个完整准确的网络拓扑图。这一点非常重要，因为每个路由器都要根据这个网络拓扑图来计算自己的路由表。正确的路由表能够提高 IP 数据包正确到达目的地的几率，而不正确或不完整的路由表则使得路由器无法将数据包转发到目的地，更严重的情况是它可能在网络上循环一段较长的时间，白白的消耗了网络带宽和路由器的资源。

路由协议可以分为域内和域间两类。一个域通常又可以称之为自治系统 AS（Autonommous System）。AS 是一个由单一实体控制和管理的路由器集合，采用一个惟一的 AS 号来标识。域内协议（又称为内部网关协议 IGP）用于在同一个 AS 中的路由器之间，其作用是计算 AS 中的任意两个网络之间的最快或者费用最低的通路，以达到最佳的网络性能。域间协议又称为外部网关协议 EGP，它用于在不同的自治系统间的路由器之间，其作用是计算那些需要穿越不同自治系统的通路。由于这些自治系统是由不同的组织来管理的，因此在选择穿越自治系统的通路时，我们所依据的标准就不只是性能了，而是要依据多种策略和标准，如费用、可用性、性能、AS 间的商业关系等。

另外根据路由协议采用的路由算法不同可以将其分为基于距离向量算法的路由协议和基于链路状态算法的路由协议。下面来看看这两种路由算法的原理。

6.1.2.1　距离向量路由算法

距离向量路由算法的基本思想是：所有路由器都会把它所知道的（不管是自己产生的还是从其他地方获得的）网络和到达该网络的距离等方面的信息告诉与其相邻的路由器。允许距离向量路由协议的路由器会向与之直接相邻的路由器发送多个距离向量，一个距离向量由一个二元组{网络地址，距离}表示。在这个二元组中，距离表示的是发送该向量的路由器与目的网络之间通路上的路由器的数量，即跳数。通过相邻路由器之间的这种距离向量的交换，

每个路由器最后都会知道通过各个端口所能到达的网络及到达该网络所需的代价。

路由器根据收到距离向量获得到达各个网络的代价的具体过程如下所述。路由器在收到相邻路由器发送的距离向量时，会把自身的距离值（通常为 1）加到它所收到的距离向量的距离值上。然后该路由器把这个新计算出来的到达目的网络的距离值与其自身的记录进行比较（如果没有该目的网络地址相匹配的记录就直接将距离向量添加到本地距离向量库中），如果新的距离值比已有的小，那么路由器就用新的距离值来更新距离向量库，并据此生成一个新的路由表，在路由表中把发送该距离向量的路由器作为达到该目的网络的转发下一站。然后路由器再将自己更新后的距离向量发送到相邻的路由器。

距离向量路由算法很简单，实现也很容易。但它有一个很严重的问题。我们看看图 6.2 所示的例子。

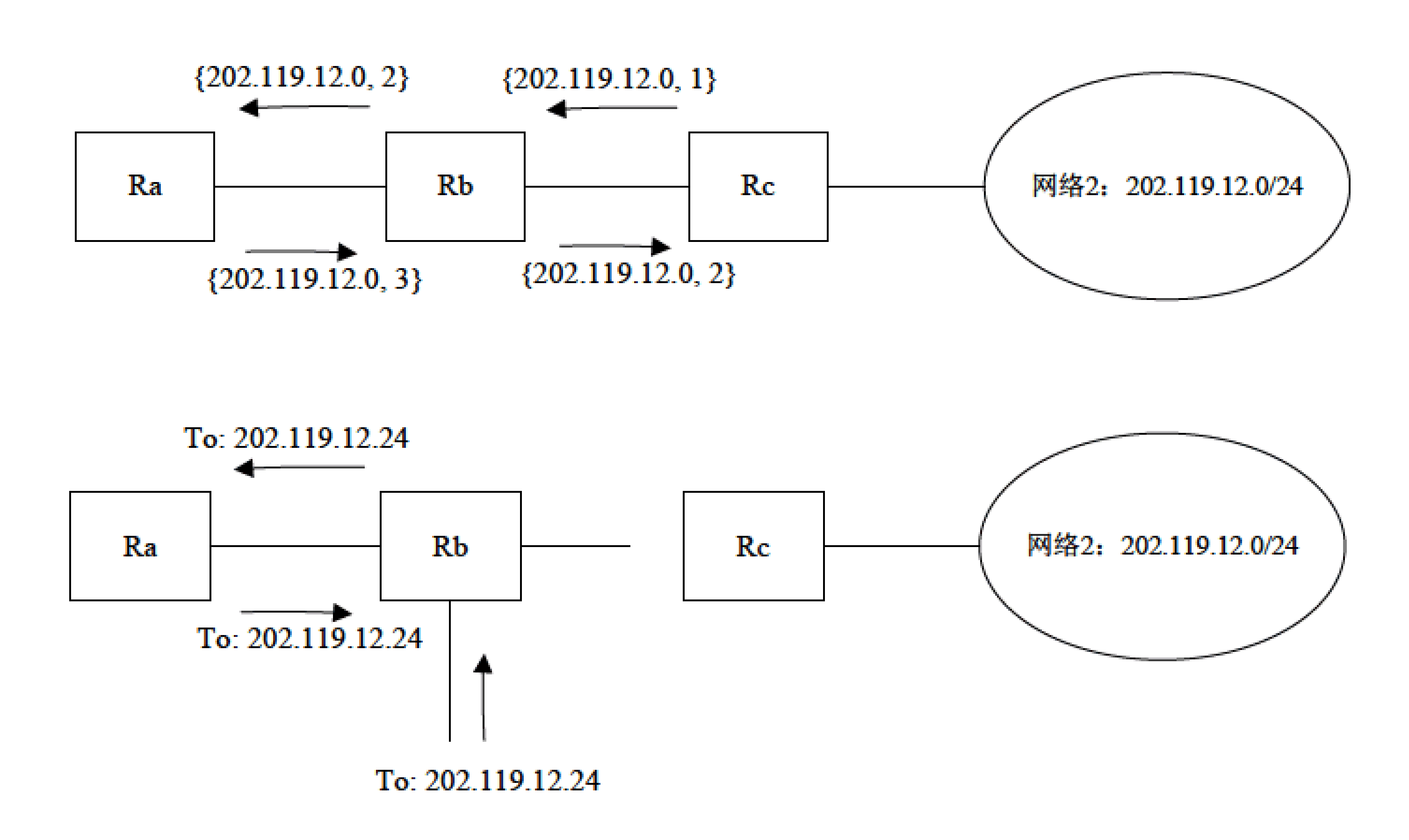


图 6.2　距离向量路由算法举例

在图 6.2 中，有 3 个路由器，其中路由器 Rc 连接到网络 202.119.12.0/24。图中上面的部分说明的是路由器之间交换距离向量信息的过程。下面部分说明的是在路由器 Rb 和 Rc 之间的通路断开之后数据包转发的情况，我们来解释一下，因为距离向量路由算法是以距离最小的向量为基础来计算路由表的，所以在图的上半部分，Rb 会将到达网络 202.119.12.0/24 的下一站设为路由器 Rc。在下半部分，Rb 则会将到达网络 202.119.12.0/24 的下一站设为路由器 Ra，因为 Ra 向它发送了向量{202.119.12.0, 3}。而 Ra 则会把达到网络 202.119.12.0/24 的下一站设为 Rb。因此当有一个数据包要发送到 202.119.12.24 时，该数据包就会在 Ra 和 Rb 之间循环。当然，现在很多路由器都可以防止从一个端口进入的数据包再从该端口发送出去，这样就能防止图 6.2 所发生的循环了。但应该清楚，这是发生循环的最简单的一个例子。如果三个或三个以上的路由器之间形成这种转发的循环则使用这种简单的措施是无能为力的。

解决这个问题的办法很简单，就是禁止把到达目的网络的距离向量发送给路由表中通往该网络的下一站。那么在图 6.2 的上半部分中，Ra 就不能发送距离向量给 Rb，Rb 也不能发送距离向量给 Rc 了。这样就不会出现循环了。

6.1.2.2　链路状态路由算法

链路状态路由算法的基本思想是：一个路由器能够把有关连接到该路由器的链路的状

态、费用及任何连接到该链路的路由器的标识等信息通知给网络中的所有其他路由器。运行链路状态路由协议的路由器会向整个网络发送链路状态数据包 LSP。一个 LSP 通常包含一个源路由器标识、一个相邻路由器标识及两者之间的链路的费用。LSP 被所有的路由器接收，用于建立一个网络的整体链路状态库，并据此得出网络的整体拓扑图，再计算出路由器的路由表。路由器在获得整个网络拓扑图后计算路由表的方法是：路由器采用最短路径或最低费用的通路建立一个以本路由器为根的路由器树。除了与本路由器直接相连的网络之外，所有网络地址的下一站都为在该路由器树上从根节点到该网络的路径上的除根节点外的最高的节点路由器。与距离向量路由算法相比，链路状态路由算法具有以下优点：

（1）更快的收敛速度。所谓收敛就是指在网络拓扑发生变化时，路由器将该变化通知给网络中所有路由器的过程。该过程越快，那么在网络拓扑发生变化时，可能发生的转发错误就更少。

（2）更小的网络开销。链路状态路由协议传送的 LSP 只反映网络拓扑的变化，而不是传送整个路由数据库。

（3）扩展性更好。

（4）更容易升级。

6.2　RIP 路由信息协议

按照对路由协议的分类，RIP 协议属于域内基于距离向量算法的路由协议。本节将具体介绍 RIP 协议的工作原理。

6.2.1　RIP 协议数据包的格式

RIP 协议数据是使用 UDP 协议进行传输的，图 6.3 说明 RIP 协议数据的封装关系。

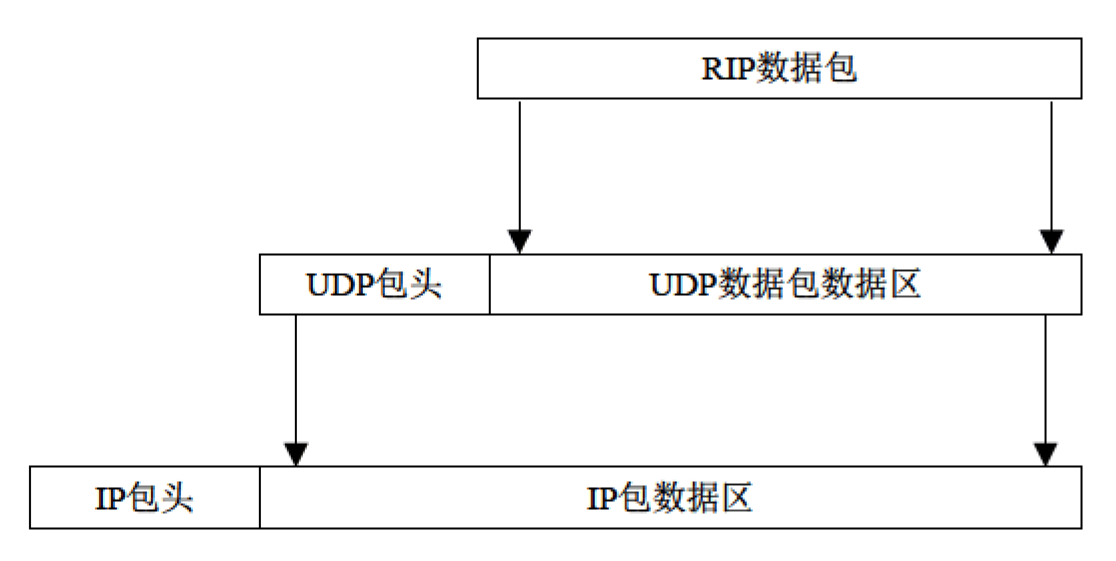


图 6.3　RIP 协议数据的传输封装

RIP 协议使用的标准 UDP 端口号为 520。RIP 协议数据包的内部格式如图 6.4 所示。

数据包的第一个字节是命令字段，其值可以是 1～6，分别为 1（请求）、2（应答）、3 和 4（不用）、5（轮询）和 6（轮询表项）。版本字段可以为 1 或 2，这里的包格式是版本 1 的格式。地址类型字段对于 IP 协议而言为 2。度量字段的单位为跳数，直接连接的度量为 1，这和前面距离向量路由算法中的一样。一个 RIP 协议数据包最多可以携带 25 条距离向量信息。

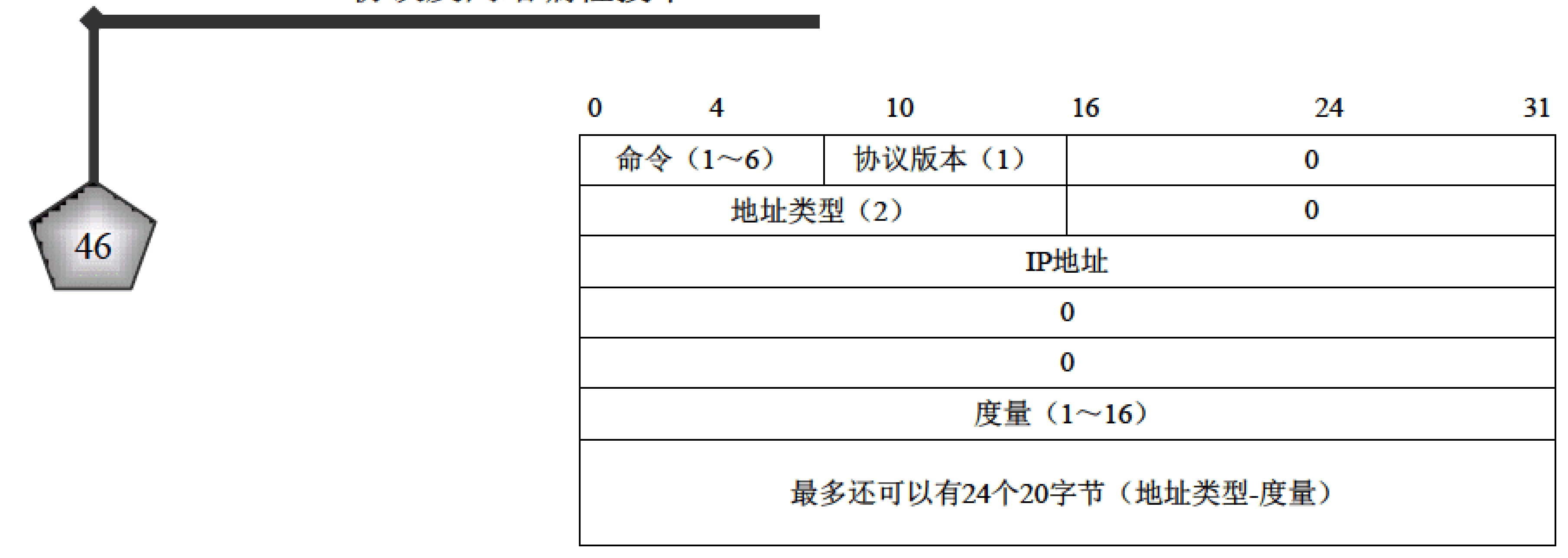


图 6.4　RIP 数据包格式

6.2.2　RIP 协议的工作过程

我们根据 RIP 协议的各个不同时间阶段以及对不同的 RIP 数据包的响应来看看 RIP 协议的工作过程。

初始化：RIP 协议启动时，它先判断路由器启动了哪些接口，并在每个接口上发送一个请求数据包，要求其他路由器发送完整路由表。在点对点链路中，该请求是发送给其他终点的。如果网络支持广播的话，这种请求是以广播形式发送的。目的 UDP 端口号是 520。这种请求数据包的命令字段为 1，但地址系列字段设置为 0，而度量字段设置为 16。这是一种要求另一端完整路由表的特殊请求数据包。

接收到请求：如果这个请求是刚才提到的特殊请求，那么路由器就将完整的路由表发送给请求者。否则，就处理请求中的每一个表项：如果有连接到指明地址的路由，则将度量设置成自己的值，否则将度量置为 16（度量为 16 是一种称为“无穷大”的特殊值，它意味着没有到达目的的路由），然后发回响应。

接收到响应：使响应生效，可能会更新路由表。可能会增加新表项，对已有的表项进行修改，或是将已有表项删除。

定期路由更新：每过 30 秒，所有或部分路由器会将其完整路由表发送给相邻路由器。发送路由表可以是广播形式的（如在以太网上），或是发送给点对点链路的其他终点的。

触发更新：每当一条路由的度量发生变化时，就对它进行更新。不需要发送完整路由表，而只需要发送那些发生变化的表项。每条路由都有与之相关的定时器。如果运行 RIP 的系统发现一条路由在 3 分钟内未更新，就将该路由的度量设置成无穷大（16），并标注为删除。这意味着已经在 6 个 30 秒更新时间里没收到通告该路由的路由器的更新了。再过 60 秒，将从本地路由表中删除该路由，以保证该路由的失效已被传播开。

6.2.3　RIP 协议的缺陷

RIP 协议看起来很简单，但它有一些缺陷。首先，RIP 没有子网地址的概念。因此无法使用子网划分技术来节省 IP 地址资源。有一些实现中通过接收到的 RIP 信息，来使用接口的网络掩码，而这有可能出错。其次，在路由器或链路发生故障后，需要很长的一段时间才能稳定下来。这段时间通常需要几分钟。在这段建立时间里，可能会发生路由环路。在实现 RIP 时，必须采用很多微妙的措施来防止路由环路的出现，并使其尽快建立。采用跳数作为

路由度量忽略了其他一些应该考虑的因素。同时，度量最大值为 15 则限制了可以使用 RIP 的网络的大小。

6.2.4　RIP2

RIP2 是 RIP 协议的第二个版本，它对 RIP 协议做了扩充但可以和 RIP 版本 1 兼容。图 6.5 是 RIP2 的数据包格式。

0　　4	10	16　　24　　31
命令（1～6）	协议版本（1）	路由域
地址类型（2）		路由标志
IP地址		
子网掩码		
下一站地址		
度量（1～16）		
最多还可以有24个20字节（地址类型-度量）		

图 6.5　RIP2 的数据包格式

RIP2 并没有改变 RIP 的数据包格式，只是将原来设为 0 的字段赋予了一定的意义。

路由域是 RIP2 协议实例的标识，它指出了这个数据包的所有者，即发送者。它允许在一个路由器中允许多个 RIP2 协议的实例。路由标记是为了支持外部网关协议而存在的。它携带着一个 EGP 或 BGP 的自治系统 AS 号。

每个表项的子网掩码应用于相应的 IP 地址上。下一站 IP 地址指明发往目的 IP 地址的数据包该发往哪里。该字段为 0 意味着发往目的地址的数据包应该发给发送 RIP 数据包的系统。

RIP2 提供了一种简单的鉴别机制。可以指定 RIP 数据包的前 20 字节表项地址系列为 0xffff，路由标记为 2。表项中的其余 16 字节包含一个明文口令。最后，RIP2 除了广播外，还支持多播，这可以减少不收听 RIP2 数据包的主机的负担。

6.3　OSPF 开放最短路径优先

OSPF 是除 RIP 外的另一个内部网关协议。和 RIP 不同的是它使用的是链路状态路由算法。它克服了 RIP 的所有限制。OSPF 与 RIP（以及其他路由协议）的另一个不同点在于，OSPF 直接使用 IP 协议，而不是 UDP，如图 6.6 所示。

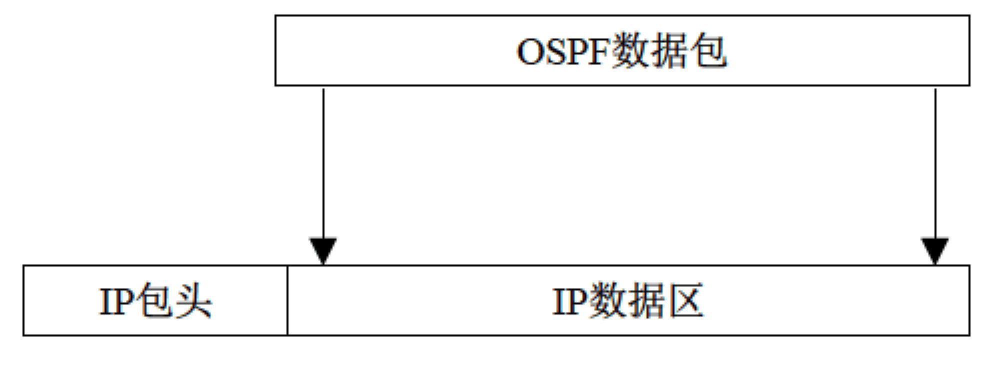


图 6.6　OSPF 数据包的封装

对于 IP 头部的协议类型字段，OSPF 有其自己的值。

在 OSPF 中，使用了一个称之为区域（Area）的概念。区域用来定义一个自治系统中的路由器和网络的集合。在 OSPF 网络中，必须有一个区域 0 用来标识网络骨干区域。如果配置了多个区域，那么所有的非 0 区域都必须通过一个区域边界路由器（ABR）连接到区域 0。在一个区域中，路由器相互发布和交换链路状态通告 LSA，并为该区域建立一个统一的映射图，称为链路状态数据库。区域之间通过 ABR 相互传递有关某一特定网络和拓扑的概括信息。因而路由器可以保存有关其所有区域中的所有网络及路由器的完整信息，以及有关在区域外网络及路由器的特殊信息。路由器中有足够的信息来引导分组通过合适的区域边界路由器到达另一个区域中的网络。

作为一种链路状态协议而不是距离向量协议，OSPF 还有着一些优于 RIP 的特点。

（1）OSPF 可以对每个 IP 服务类型计算各自的路由集。这意味着对于任何目的，可以有多个路由表表项，每个表项对应着一个 IP 服务类型。

（2）给每个接口指派一个无维数的费用。可以通过吞吐率、往返时间、可靠性或其他性能来进行指派。可以给每个 IP 服务类型指派一个单独的费用。

（3）当对同一个目的地址存在着多个相同费用的路由时，OSPF 在这些路由上平均分配流量，称之为流量平衡。

（4）OSPF 支持子网，子网掩码与每个通告路由相连。这样就允许将一个任何类型的 IP 地址分割成多个不同大小的子网。到一个主机的路由是通过全 1 子网掩码进行通告的。默认路由是以 IP 地址为 0.0.0.0、网络掩码为全 0 进行通告的。

（5）路由器之间的点对点链路不需要每端都有一个 IP 地址，称之为无编号网络。这样可以节省 IP 地址——现在非常紧缺的一种资源。

（6）采用了一种简单鉴别机制。可以采用类似于 RIP2 机制的方法指定一个明文口令。

（7）OSPF 采用多播，而不是广播形式，以减少不参与 OSPF 的系统负担。

随着大部分厂商支持 OSPF，在很多网络中 OSPF 将逐步取代 RIP。

6.4 BGP 边界网关协议

BGP 是一种不同自治系统的路由器之间进行通信的外部网关协议。BGP 是 ARPANET 所使用的老 EGP 的取代品。BGP 的主要目标是为处于不同 AS 中的路由器之间进行路由信息通信提供保证。BGP 在发送一个目的网络的可达性的同时会包含 IP 数据包到达目的网络过程中必须经过的 AS 的列表，这个列表称为通路向量。通路向量信息是很有用的，因为只要简单的查找以下 BGP 路由更新中的 AS 编号即能有效的避免环路的出现。

BGP 系统与其他 BGP 系统之间交换网络可到达信息。这些信息包括数据到达这些网络所必须经过的自治系统 AS 中的所有路径。这些信息足以构造一幅自治系统连接图。然后，可以根据连接图删除环路，制订路由策略。首先，将一个自治系统中的 IP 数据包分成本地流量和通过流量。在自治系统中，本地流量是起始或终止于该自治系统的流量。也就是说，其信源 IP 地址或目的 IP 地址所指定的主机位于该自治系统中。其他的流量则称为通过流量。在 Internet 中使用 BGP 的一个目的就是减少通过流量。

可以将自治系统分为以下几种类型：

（1）残桩自治系统（stub AS），它与其他自治系统只有单个连接。stub AS 只有本地流量。

（2）多宿主自治系统（multihomed AS），它与其他自治系统有多个连接，但拒绝传送通过流量。

（3）转送自治系统（transit AS），它与其他自治系统有多个连接，在一些策略准则之下，它可以传送本地流量和通过流量。

这样，可以将 Internet 的总拓扑结构看成是由一些残桩自治系统、多接口自治系统以及转送自治系统的任意互连。残桩自治系统和多接口自治系统不需要使用 BGP——它们通过运行 EGP 在自治系统之间交换可到达信息。BGP 允许使用基于策略的路由。由自治系统管理员制订策略，并通过配置文件将策略指定给 BGP。制订策略并不是协议的一部分，但指定策略允许 BGP 实现在存在多个可路由径时选择路径，并控制信息的重发送。路由策略与政治、安全或经济因素有关。

BGP 与 RIP 和 OSPF 的不同之处在于 BGP 使用 TCP 作为其传输层协议。两个运行 BGP 的系统之间建立一条 TCP 连接，然后交换整个 BGP 路由表。从这个时候开始，在路由表发生变化时，再发送更新信号。BGP 是一个距离向量协议，但是与 RIP 不同的是，BGP 列举了到每个目的地址的路由（自治系统到达目的地址的序列号）。这样就排除了一些距离向量协议的问题。采用 16 位数字表示自治系统标识。BGP 通过定期发送 keepalive 数据包给其邻站来检测 TCP 连接对端的链路或主机失败。两个数据包之间的时间间隔建议值为 30 秒。

6.5　Internet 的路由体系结构

上面介绍了各种路由协议的机制，本节要说明这些不同的路由协议是怎样存在于整个 Internet 中的。图 6.7 就说明了这种共存关系。

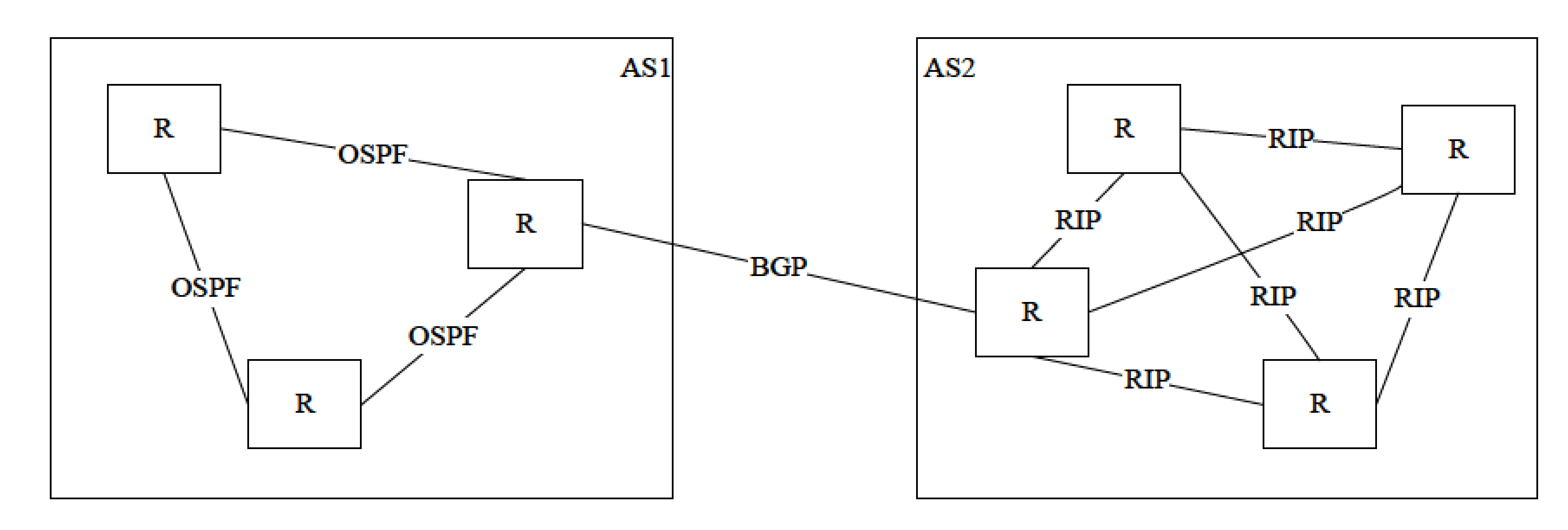


图 6.7　Internet 中可以同时允许多种路由协议

从图中可以看出，在 AS 内部允许的域内路由协议，而在 AS 的边界路由器之间允许运行的是域间路由协议。当然，这只是一个示意图，实际上路由协议的关系远比图中所示复杂，它甚至允许在同一个 AS 内部允许多种路由协议。

第 7 章　广播与多播

到目前为止，我们讲的技术都是点到点的数据传输技术（单播）。即使有的技术中要使用到一点到多点的传输技术（如 ARP、OSPF 等），也没有深入研究这种一点到多点的传输技术是怎样工作和实现的。本章开始讨论这个问题。因为一点到多点的数据传输可以分为广播和多播两种，而广播技术比多播简单得多，所以先用一节说明广播的工作机制，后面几节用来解释多播技术。

7.1　广　　播

所谓广播就是发送一个数据给一个范围内（通常是一个网络）的所有系统，该范围内的所有系统都应该接收该数据。使用 IP 协议可以实现数据广播。和单播的数据包一样，广播的数据包也是封装在物理网络的数据帧中进行传输的。

我们前面讲到过，数据包或数据帧中必须要有数据的目标系统的地址。由于广播数据包实际是有多个接收者的，有几个原因使得发送无法将每个接收者的地址都放入数据包或数据帧中。第一，数据包和数据帧中只有一个目的地址字段，无法容纳多个目的地址；第二，发送者往往也不知道数据的确切接收者都是谁。例如要向网络内的所有主机发送数据并不需要知道网络中都有哪些主机，它们的地址是什么。

因此，必须有一种特殊的地址，它能代表网络中的所有主机。这种地址就称之为广播地址。不同的协议层和同一层次中的不同技术使用的技术都是不同的。

7.1.1　物理层的广播

由于 IP 数据包都是封装在物理网络层的数据帧中进行传输的，接收方的 IP 也只有在底层的物理网络接收了数据帧之后才能从物理网络层协议取得该数据包。所以 IP 层要将数据进行广播，必须通知物理网络将数据包进行广播。所以物理层也需要有广播地址的表示方法，要有广播数据的发送和接收技术。

大多数物理网络技术都有表示广播地址的方法。例如在以太网中，每个以太网地址长 48 位，所有位都为 1 的地址就表示广播地址。以太网的网卡可以接收两种数据帧，一种就是目的地址是本网卡物理地址的数据帧，另一种就是目的地址是广播地址的数据帧。

物理网络中广播数据帧的发送技术也各不相同。例如在以太网这样的总线型广播式网络中，广播可以通过发送一个数据帧实现，因为所有网络中的网卡都会收到该数据帧。而在一些点到点连接的非广播型网络中，要发送一个广播数据帧必须向网络中每个系统单独发送一个数据帧。

7.1.2　IP 协议的广播

在第 3 章中详细介绍了各种 IP 地址，在 IP 地址的五种类型中并没有广播地址类型。IP 地址由两部分组成，即网络标识和主机标识。IP 协议的广播地址就是将网络标识或主机标识设置为特定的值来表示的。

首先，有一个特别的 IP 地址：255.255.255.255。这个地址用来表示本网络内广播。路由器接收到目的地址为该地址的数据包是不会将其进行转发的。因此使用该地址的广播数据包是不会扩散到本地网络之外的。这个地址对于 IP 地址自动分配技术很有用，因为系统在启动根本无法知道本地网络的标识。它就是用该地址发送一个广播数据包并等待地址分配服务器的响应。

另外，主机标识各位都为 1 的地址表示指定网络的广播，这个地址中的网络标识部分指定了该广播数据包应该在其中进行广播的网络，该网络可以不是本地网络。

7.1.3　IP 广播的过程和问题

当 IP 要发送一个广播数据包时，首先它要确定这个广播是本地网络内的广播还是其他网络的广播。第一种情况，如果是本网络内的广播，它应该将目的 IP 地址设为第一种 IP 广播地址或网络标识为本地网络标识的第二种 IP 广播地址，然后要求底层物理网络将该数据包广播出去。第二种情况，如果是外部网络的广播，它就应该将目的 IP 地址设为第二种 IP 广播地址，但不能要求物理网络广播该数据包。而是将数据包以单播形式发送给路由器。路由器在处理这种广播数据包时，先与单播数据包一样查找路由表确定下一站。如果下一站是另一个路由器，那它的处理方式就和单播数据包一样，将它转发给下一站路由器。如果发现下一站是直接连接，即目的网络是直接连接在该路由器上的，这时路由器就要求相应端口的物理网络层以广播的形式将数据发送到相应的网络上，同时路由器还需要将该数据包提交给路由器的相应上层协议处理，因为路由器也是该网络中的一个系统。

对于广播包的接收和单播包的接收过程相同。

需要强调的是：广播地址只能用作目的地址，数据包或数据帧中的源地址决不允许使用广播地址。

广播技术虽然很简单，有时也很有用，但它消耗的资源太多，所以应尽量限制它的使用。因为广播网络中的每个系统的各层协议（从物理层知道该数据的最终目标层）都需要处理该数据包，而其中大多数系统最终还是将其丢弃。

7.2　多　　播

广播中，接收方是被动的一方。因为不管它愿不愿意，它都必须接收广播数据包并对其进行处理，尽管最后还是将其丢弃。多播也是一种将一个数据包发送给一个范围内的所有系统的技术，但和广播技术不同的是，接收方系统可以选择是否加入到这个范围中，即它可以选择是否接收一个多播数据包。这样既实现广播的方便性，又不至于浪费系统的资源。

但实现多播的技术比广播复杂的多。首先，在多播中，参与多播的接收者的数量和范围是不固定的，因为接收者可以选择是否加入。我们将参与多播的所有接收者称为组，所以多播也可以称为组播。多播对于一些应用是很重要的，例如电视会议、聊天室等。但是由于组中成员是不固定的，甚至可以在组建立之后加入或退出。所以多播的地址不能使用广播的那种固定的表示方法。另外，同一时间可能会存在多个组，而同一个系统也可以同时加入多个组。这也增加多播地址的复杂性。下面具体看看多播地址和技术的具体机制。

7.2.1　物理层的多播

所有的数据传输和接收都需要物理层的参与，多播技术也一样，所以物理层也需要有多播的表达的传输技术。

首先还是目的地址的表示问题。 多数物理网络技术都保留了一部分地址空间用来表示多播地址。和广播技术只需要一个或几个广播地址不同，因为在同一个网络可能同时会存在多个多播组，为了区分不同的组，所以需要有较多的多播地址。当一组系统需要进行多播通信时，它们可以从这些保留的地址空间中选取一个地址所谓该组的多播地址。为了接收该目的地址为该多播地址的数据帧，每个系统都必须对它的网络接口进行配置，让它识别该地址并接收目的地址为该地址的数据。

各种物理网络由于地址的表示不同，所以多播地址的表示也不同。我们以以太网为例说明多播地址技术。在以太网地址中，最高字节的最低位为 1 的地址都是多播地址。例如：43:24:D4:54:37:01 就是一个多播地址，而广播地址（FF:FF:FF:FF:FF:FF）是多播地址的一个特例。

以太网中，网卡的初始设置只会接收物理地址为本网卡地址和广播地址的数据帧。但可用通过简单设置使它接收特定多播地址的数据帧。

7.2.2　IP 协议的多播

IP 协议的多播除了具有上述的组成员的动态加入和退出特点外，还有一个优点，即它的组成员可以是跨网络的，没有范围的限制。IP 协议的多播数据包可以通过物理网络的多播技术在本地网络中进行传输，但由于物理网络的多播技术无法跨越网络，所以 IP 协议需要特定的技术来将 IP 数据包进行传输。要将多播数据包进行跨网络的传送，必须有一种特殊的路由器的帮助，称为多播路由器。我们后面讨论具体的工作过程。

7.2.2.1　IP 多播地址

IP 协议的五种地址类型中，D 类地址是专门用作进行多播通信的。D 类地址的地址范围是 224.0.0.0～239.255.255.255。其中 224.0.0.0 被保留不能赋给任何多播组，224.0.0.1 是一个所有主机组地址，它表示参与 IP 多播的所有主机和路由器。通常，所有主机组地址用来表示本地网络中所有参与 IP 多播的主机。IP 协议中没有代表整个 Internet 中所有主机的多播　地址。

需要注意的是，IP 多播地址和广播地址一样，只能用作目的地址而不能用在 IP 数据包头的源地址字段中。它们也不能出现在源路由和记录路由选项中。另外，不能为多播数据包

产生 ICMP 错误信息。

7.2.2.2　IP 多播地址到以太网多播地址的映射

因为 IP 多播数据包最终是要以物理层网络的多播数据帧进行传输的，所以使用 IP 多播地址来指定一个多播组地址后，IP 协议还必须告诉物理网络应该将该数据包发送到哪个物理层的多播组中。

IP 协议标准中指定了一种将 IP 多播地址映射为以太网多播地址的方法。这个方法很简单：要将 IP 多播地址转换为一个相应的以太网多播地址，只需将 IP 多播地址的低 23 位填入以太网地址 01:00:5E:00:00:00 的低 23 位即可。例如 IP 多播地址 224.0.0.1 就映射到以太网多播地址 01:00:5E:00:00:01。

需要指出的是，这种映射不是惟一的。因为 IP 地址的 D 类地址中，除了最高的 4 位为 D 类地址的标识外，还有 28 位是可变的。将 28 位映射为 23 位就意味着 32 个 IP 多播地址共用一个以太网的多播地址。虽然这种映射不是惟一的，但因为 IP 多播地址的范围很大，主机在选择多播 IP 地址时，选中两个低 23 位相同的地址的可能性是很小的。但因为可能会存在重复的映射，所以 IP 协议对收到的多播数据包必须小心处理，将不是发往主机的数据包丢弃。

7.2.2.3　IP 协议对多播数据包的处理

主机要发送一个多播数据包很简单。IP 协议软件应该允许应用程序将一个多播地址指定为数据的目的地址。物理网络层的接口也应该能将一个 IP 多播地址映射为一个物理多播地址。这样多播数据包就可以发送出去了。

对于多播数据包的接收，则比发送过程复杂。首先，IP 协议软件需要提供一个接口，允许用户加入一个多播组。同一个主机中可能有多个应用程序加入了同一个多播组。如果这样，IP 协议必须给每一个应用程序复制一份该数据包的副本。如果所有的应用程序都退出了一个多播组，那么 IP 协议就不应该再接收该多播组的数据包。

对 IP 多播数据包的路由就更复杂了，这需要多播组中主机系统和多播路由器的合作才能完成。在 7.3 节将讨论 IP 多播数据包的路由问题。

7.3　IGMP

要加入本地网络中的多播通信组，主机只需要能发送和接收多播数据包即可。但要加入一个跨网络的多播通信组，主机还需要将字节加入的信息通知给本地的多播路由器。本地路由器与其他的多播路由器联系，传递多播组成员信息并建立路由。而在多播路由器传递多播组成员信息给其他多播路由器之前，它必须确定在本地网络中有主机系统加入了该多播组。主机和多播路由器是通过 Internet 组管理协议 IGMP（Internet Group Management Protocol）来管理多播组测成员关系的。

7.3.1　IGMP 数据包格式

IGMP 数据包的长度是固定的，为 8 个字节，即 64 位。其结构如图 7.1 所示。

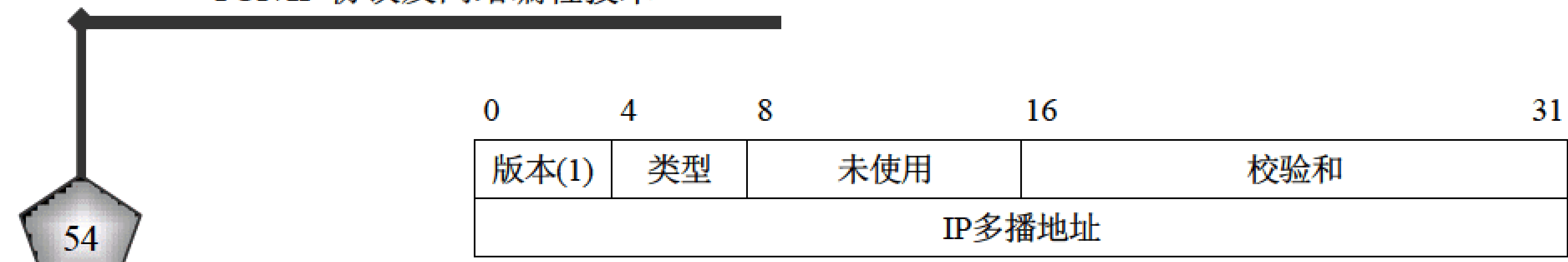


图 7.1　IGMP 数据包格式

类型字段的值可以为 1 或 2。1 表示是由多播路由器发出的查询数据包，2 表示是主机发出的报告数据包。校验和的计算和 ICMP 协议相同。组地址为 D 类 IP 地址。在查询数据包中组地址设置为 0，在报告数据包中组地址为要参加的组地址。

和 ICMP 一样，虽然 IGMP 是封装在 IP 数据包中进行传输的，但它被当作 IP 层的一部分。IP 头部协议类型字段用 2 表示 IGMP 协议。

7.3.2　IGMP 协议的工作机制

IGMP 协议的整个工作过程可以分为两个部分。

第一部分，当一个主机加入一个新的多播组时，它发送一个 IGMP 报告数据包到网络 上。该数据包的 IP 多播地址字段就设置为它加入的多播地址，且该数据包的目的地址设为所有主机多播地址 224.0.0.1。因为使用这个目的地址的数据包会被本地网络中所有参与多播的主机和路由器所接收，所以主机不需要知道本地多播路由器的 IP 地址。多播路由器在接收到该数据包后会通过向其他多播路由器传播组成员信息来建立必要的路由信息。

第二部分，因为多播组的成员是变化的，随时可能有主机加入一个组，也随时有主机退出一个组。所以，本地的多播路由器需要定期的查询每个组中在本地网络还有那些成员。这是通过向网络中发送 IGMP 查询数据包完成的。多播路由器向每个端口连接的网络中发送一个目的地址为 224.0.0.1 的 IGMP 查询数据包，在数据包中 IP 多播地址字段设置为 0。每个参与多播通信的主机在收到 IGMP 查询数据包后，都需要发送一个上面所述的 IGMP 报告数据包，IP 多播地址字段和目的地址都设为其参与的多播组的地址。如果一个主机加入多个多播组，那么主机需要为每个组发送一个报告数据包。

7.3.3　IGMP 协议的实现

所有的 IGMP 数据包都是以多播的形式发送的，因此实现时必须尽量减少数据包的发送以减少对网络造成的影响。为改善该协议的效率，有许多实现的细节要考虑。

首先，当一个主机加入一个多播组而首次发送 IGMP 报告时，并不能保证该报告被可靠接收。下一个报告将在间隔一段时间后发送。这个时间间隔由主机在 0～10 秒的范围内随机选择。如果多个应用程序加入同一个多播组，主机不会为每一个加入的应用程序发送一个报告，只会在第一个应用程序加入时发送报告。

其次，当一个主机收到一个从路由器发出的查询后，并不立即响应，而是经过一定的时间间隔后才发出响应，这个延迟的时间是随机的。在一个物理网络中的所有主机将收到同组其他主机发送的所有报告，因为报告中的目的地址是那个组地址。这意味着如果一个主机在等待发送报告的过程中，却收到了发自其他主机的相同组的报告，则该主机的响应就可以不必发送了。这是因为多播路由器并不关心有多少主机属于该组，而只关心该组是否还至少拥

有一个主机。的确，一个多播路由器甚至不关心哪个主机属于一个多播组，它仅仅想知道在该网络中的多播组中是否还至少有一个主机。在没有任何多播路由器的本地物理网络中，仅有的 IGMP 通信量就是在主机加入一个新的多播组时，支持多播的主机所发出的报告。

IGMP 报告和查询数据包的 TTL 值设置为 1。一个初始 TTL 为 0 的多播数据包将被限制在同一主机。在默认情况下，待传多播数据包的 TTL 被设置为 1，这将使多播数据包仅局限在同一子网内传送。更大的 TTL 值能被多播路由器转发。对发往一个多播地址的数据包从不会产生 ICMP 差错。

第 8 章　UDP 协议

UDP 协议和 IP 协议一样，也提供不可靠的传递服务。那么为什么还需要 UDP 协议呢？我们知道 IP 协议中数据包的最终地址是一个 IP 地址，也即代表了一个主机。但在多任务的操作系统中可能同时会有多个任务（即应用程序）在执行，它们可能都需要进行网络通信。这时就出现了一个问题，即收到 IP 数据包的主机系统的 IP 协议应该将该数据包交给哪个应用程序进行处理呢？另外，在第 3 章讲到，IP 协议只对 IP 头部计算校验和以保证头部在传输过程中不被改动，那么数据部分的完整性由谁来保证呢？所有这些都决定了只依靠 IP 协议进行数据传递是不够的。

8.1　最终目标的标识——UDP 端口

上面讲到在多任务的操作系统中会有多个应用程序同时在执行，而仅仅依靠 IP 协议的 IP 地址无法区分一个主机系统中的多个应用程序。因此需要有一种机制，使得数据包的发送方能够指定该数据不是发送给目标主机中的哪一个应用程序的。

最直接的一种方法就是在指定 IP 地址的同时还指定接收该数据包的应用程序的进程 id。这样在接收主机上的 IP 协议就知道应该将该数据包交给哪个应用程序处理了。但这种方法有几个致命的缺点它是不可做到的。第一，在大多数操作系统中，进程的建立与销毁是动态的，而进程 id 的分配也是动态的。那么同一个应用程序在不同的时间其进程 id 也有可能是不同的。因此，发送方根本就无法知道他要与其通信的进程的 id 是什么；第二，有时需要在一个进程中实现多种功能，那么该进程就需要对接收到的数据包进行区分，以识别它是哪种功能的数据包；第三，我们可能需要访问目标主机的某种标准功能，而不需要知道实现这种功能的应用程序是什么。因为实现这种功能的应用程序有很多种，因此不可能指定该应用程序的进程 id。所以，通过指定进程 id 的方式来指定数据包的最终目标的做法是不可行的。

UDP 协议采用的最终目标的表示方法是使用 UDP 端口，每个端口就是一个最终目标。在每个主机的 TCP/IP 协议栈中，UDP 协议有一个端口集，其中每个端口都用一个整数进行标示。可以将一个端口理解为一个队列。在发送一个数据包时，通过指定该数据包的目标 IP 地址和端口号来指定它的最终目标。在接收方，UDP 协议将属于一个端口的数据包放在一个队列当中。应用程序在进行网络通信之前必须先申请一个或多个属于它自己的端口号，那么所有发往该端口的数据包都是发往该应用程序的。使用这种方法就能解决上面提到的几个问题。首先，端口号是固定的，可以为不同的功能分配固定的端口号。这样发送方应用程序就只需要将数据包发往该固定端口。其次，一个实现多个功能的应用程序可以申请多个端口，每一个端口作不同的用途。这样它就可以区分不同端口的数据包的不同用途了。最后对于标准的功能，可以预先分配固定的端口号，这些端口号是不能作为其他作用的，只有实现了该

标准功能的应用程序可以申请该端口号，那么访问该功能的应用程序就可以通过将数据包发往该端口来访问该功能了。

8.2　UDP 数据包格式

首先，需要指出的是 UDP 协议是提供不可靠传递服务的，即 UDP 协议数据包可能会丢失，失序等，UDP 协议不处理数据包的重发。这一点和 IP 协议一样。这就需要使用 UDP 协议的应用程序自己处理数据包的重发，顺序重组等。那么有的读者可能会产生疑惑，为什么不让 UDP 协议提供可靠的传递服务呢，这样不就减轻了应用程序的负担吗？这是因为要提供可靠的传递服务需要采用一些技术，这会带来额外的负担。有的应用程序可能不能负担这种负担。关于这一点将在下一章进行详细讨论。本节看看 UDP 协议数据包的格式，并解释它是怎样标示最终目标并保证 UDP 数据包的数据完整性的。

UDP 数据包的格式

UDP 数据包的格式很简单，如图 8.1 所示。

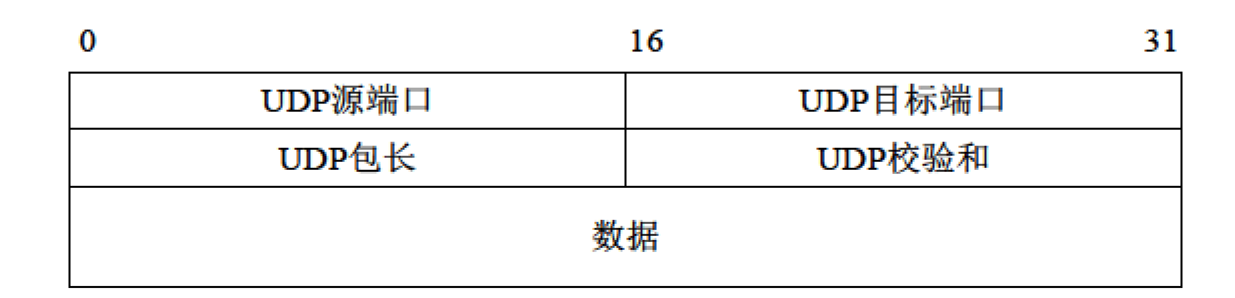


图 8.1　UDP 数据包的格式

在图 8.1 中源端口和目标端口字段指定了两个 16 长的端口号。其中源端口字段是可选的。如果指定了该字段的值，它就表示相应数据包应发往的端口号。如果不使用，应将其设为 0。长度字段表示整个 UDP 数据包的 8 位字数，包含 UDP 头部和数据部分。因此，该字段的值最小为 8。

UDP 校验和字段是用来保证 UDP 数据包的完整性的。但该字段是可选的，即 UDP 协议可以计算校验和，也可以不计算，没计算校验和的 UDP 数据包应该将校验和字段设为 0。读者可能会奇怪，为什么设置了该字段而又不使用呢？这是因为 UDP 协议的设计者考虑到在有的可靠性很高的网络中，传输的数据几乎不会出错，这样就可以通过不计算 UDP 数据包的校验和来减少主机的计算工作量。

UDP 校验和的计算方法与 IP 校验和的计算方法一样，即将数据分为 16 位长的段然后计算它们的异或。但 UDP 校验和的计算不是严格按照图 8.1 所示的数据包格式的数据包进行的，具体的计算方法在 8.3 节解释。

8.3　UDP 校验和的计算

UDP 校验和的计算不仅包含了 UDP 数据包中的所有数据还包括一个称为伪头部的结构

和将 UDP 数据包补足 16 位的整数倍的一个全为 0 的 8 位字。计算校验和时，UDP 协议先构造该数据包的一个伪头部结构，然后将 UDP 数据包的校验和字段设置为 0 并将其连接在伪头部后面，将 UDP 数据包的长度补足为 16 位的整数倍，最后按照 IP 协议校验和的计算方法对这个新的结构计算校验和并将结果填入校验和字段。UDP 伪头部和长度补足部分不会进行传输，其长度也不包含在 UDP 数据包长度字段内。

8.3.1　UDP 伪头部格式

UDP 伪头部的格式如图 8.2 所示。

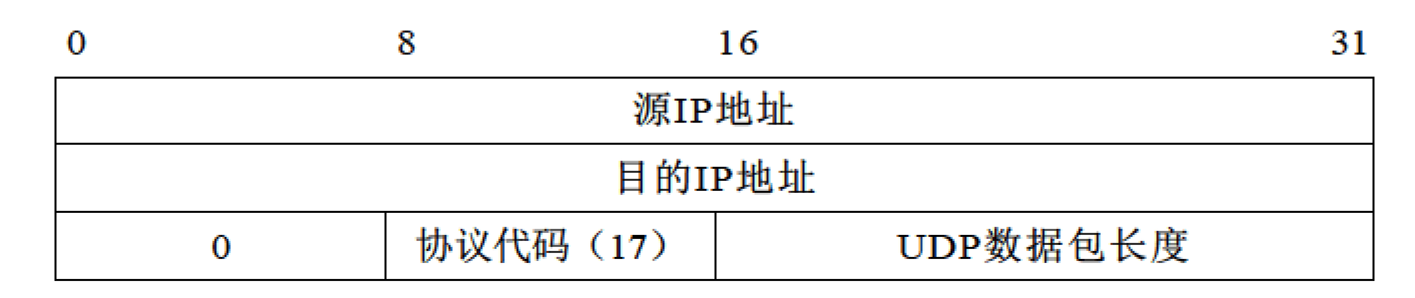


图 8.2　UDP 伪头部结构格式

源 IP 地址和目的 IP 地址字段包含了发送该数据包的源主机和接收它的目的主机的 IP 地址。协议代码字段为 UDP 协议的代码。UDP 数据包长度字段就是 UDP 数据包的 UDP 包长字段的值。

8.3.2　为什么使用伪头部

使用 UDP 伪头部的目的是为了让数据包的接收者确定发送和接收的 UDP 数据包是来自正确的源地址且该数据包是发给自己的。我们知道在 UDP 数据包的结构中只包含了数据包的源端口和目的端口，而没有包含源 IP 地址和目的 IP 地址。因此，UDP 使用伪头部结构来计算校验和。在发送方构造一个伪头部结构与待发送的 UDP 数据包一起计算校验和后发送给接收方。在接收方使用同样的方法计算校验和并与发送方计算的校验和进行比较，如果两个校验和匹配，就说明该数据包是发给本主机的，且数据传输没有出错。

8.4　UDP 数据包的封装

从 TCP/IP 协议栈的体系结构可以看到 UDP 是出于 IP 协议和应用程序之间的，图 8.3 说明了 UDP 协议在整个 TCP/IP 协议中的位置。

从图 8.3 中可以看出，UDP 协议位于 IP 协议之上。这就说明 UDP 数据包是封装在 IP 数据包中进行传输，即整个 UDP 数据包是作为 IP 数据包的数据部分被封装在 IP 数据包中的。封装关系如图 8.4 所示。

IP 数据包的头部有一个协议字段，该字段表明 IP 数据包封装的是上层哪一种协议的数据包。对于 UDP 数据包，该字段的值为 17。下面解释一个使用 UDP 协议的应用程序是如何将数据传输到目标主机 A 的特定应用程序的。

首先接收数据的应用程序要申请一个 UDP 端口号，设为 P。发送方的应用程序准备好数

据后，将其交给 UDP 协议，让其将该数据发送给主机 A 的端口 P。UDP 协议将应用程序的数据作为 UDP 数据包的数据部分封装在一个 UDP 数据包中，并将数据包的目标端口字段设置为 P。UDP 协议再将 UDP 数据包交给 IP 协议处理，让其将该数据包发送到主机 A。IP 协议将 UDP 数据包作为 IP 数据包的数据封装在一个 IP 数据包中，并将目的地址设置为 A，将协议字段设置为 17，然后将其交给网络层处理并发送出去。该 IP 数据包可能会经过数个路由器，并最终到达主机 A 的 IP 协议层。

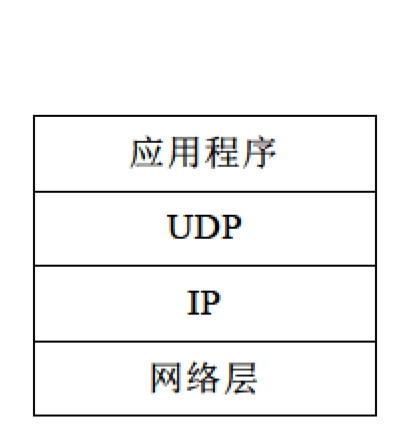


图 8.3　UDP 协议的层次图

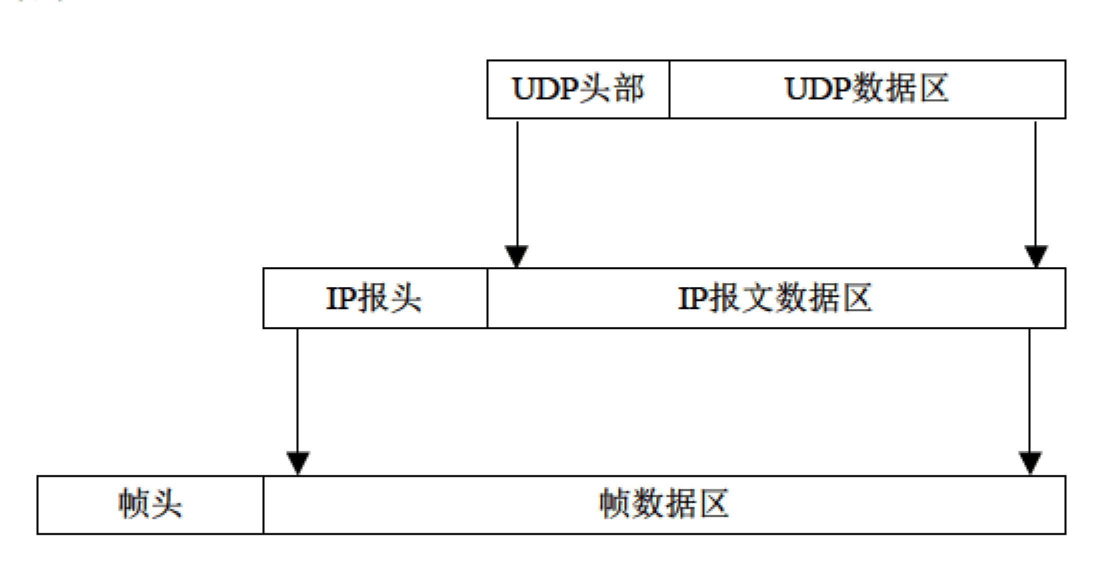


图 8.4　UDP 数据包封装关系

主机 A 的 IP 协议发现协议字段为 17，就将 IP 数据包的数据区交给 UDP 协议处理。UDP 协议发现端口号为 P，就将 UDP 数据包的数据区放置在端口 P 的队列中。A 的应用程序从该队列中将数据取出进行处理。

8.5　标准 UDP 端口

在讲述 UDP 数据包的格式时，我们注意到，尽管源端口字段是可选的，但目标端口号是必须指定。这是因为目标主机的 UDP 协议必须知道端口号才知道将数据放入哪个队列中。现在来看看一个应用程序如何才能知道要将数据发往目标主机的哪个 UDP 端口。

一种方法就是发送方在发送 UDP 数据包时指定源端口字段，应用会接收该发往端口的数据包。这样该数据包的接受者如果想发送响应数据包给该主机就可以将目标端口设置为该数据包的源端口了。但第一个 UDP 数据包的发送者如何知道数据包的目标端口呢。这可以通过为一些标准的服务指定专用的 UDP 端口实现。表 8.1 是一些常用的标准 UDP 端口及使用该端口的应用程序应提供的服务。

表5.1　标准UDP端口表

端　口　号	描　　述
0	保留
7	回显
9	丢弃
11	活动用户
13	日期时间
15	netstat
17	qotd

续表

端　口　号	描　　述
19	字符产生服务
37	时间
42	主机名服务器
43	whois
53	域名服务器
67	Bootstrap 协议服务器
68	Bootstrap 协议客户端
69	TFTP
123	网络时间协议
161	snmp
162	snmp-trap

应用程序申请 UDP 端口号可以采用两种方式。第一种就是指定需要分配哪个端口；第二种方法不指定需要的端口，操作系统可以随意分配一个可用的端口号给该应用程序。通常，如果应用程序需要接收其他主机的应用程序发出的第一个数据包，它就需要采用第一种方式申请一个固定的端口号，且这个端口号必须是其他主机的应用程序知道的。否则应用程序可以采用第二种方法申请端口号，并在发出的第一个数据包中指定源端口号。

第 9 章　TCP 协议

TCP 协议和 UDP 协议在 TCP/IP 协议栈中都处于相同的层次，位于 IP 层之上，应用层之下。但和 UDP 仅提供不可靠的传递服务不同，TCP 协议为上层应用提供面向连接的可靠的字节流传送服务。在本章将详细介绍 TCP 协议中的基本概念及 TCP 协议的工作原理，并将 UDP 协议与 TCP 协议进行比较，指出它们的不同的应用环境。

9.1　TCP 协议中的基本概念

TCP 协议是一种提供面向连接的可靠的字节流传送服务的协议。这里有几个关键的概念：面向连接、可靠的、面向字节流。下面将一一解释这几个概念。

9.1.1　面向连接的服务

和使用 UDP 协议的应用程序一样，使用 TCP 协议的应用程序也需要有一种机制指定协议数据的最终目标。同样，TCP 也使用端口的概念来标识协议数据的最终目标。TCP 端口号也是一个整数。

在第 8 章提到 UDP 的端口实际上就是一个队列，所有发往该端口的数据包都在该队列上等待同一个应用程序的处理。但在 TCP 协议中，一个 TCP 端口并不能最终决定处理 TCP 协议数据的应用程序是谁。因为 TCP 是面向连接的，所以 TCP 采用连接来标识数据的最后处理者。在 TCP 协议中一个连接由两个地址、端口对表示。例如，（192.168.8.21, 20）标识主机 192.168.8.21 上的 20 号 TCP 端口，那么（192.168.8.21, 20）-（192.168.8.22, 29）就是一个连接。从这种表示方法可以看出，一个 TCP 连接由两个端点组成。还有一点需要强调的是，TCP 的一个端口可以为多个连接复用。例如连接（192.168.8.21, 20）-（192.168.8.22, 29）和（192.168.8.37, 147）-（192.168.8.22, 29）可以同时存在。这里主机 192.168.8.22 的 29 号端口就同时成为了两个连接的端点。尽管两个连接同时使用一个 TCP 端口，但两个 TCP 连接的数据的处理程序可能并不是同一个应用程序。

从上述说明可以看出，TCP 协议是使用连接作为 TCP 数据处理程序的处理者的。而且 TCP 的连接由两个端点组成，这就决定了使用 TCP 协议传输数据只能两两之间进行通信，无法使用第 7 章中的广播和多播通信。

9.1.2　可靠的服务

从 TCP/IP 协议栈结构图可以看出，TCP 协议是位于 IP 层之上的。TCP 协议数据的传递需要依赖于 IP 协议。而我们一直都强调 IP 协议提供的是不可靠的传递服务，即 IP 数据包可

能会丢失、失序、重复等。而 TCP 要提供的服务是一种可靠的服务，即交给 TCP 协议传送的数据必须按顺序到达目标应用程序。读者可能会奇怪，在下层不可靠的网络的基础上如何能够提供可靠的传输服务呢？这是通过使用超时重发机制实现的，即接收方在收到发送方的数据后应该给发送方发送一个确认，发送方在发出数据后会等待接收方的确认，在收到确认之前不会发送下一个数据。如果等待一定的时间之后还没有收到确认就将刚发送的数据进行重发，直到收到确认或最后重发次数试完为止。图 9.1 是这种重发机制的示意图。

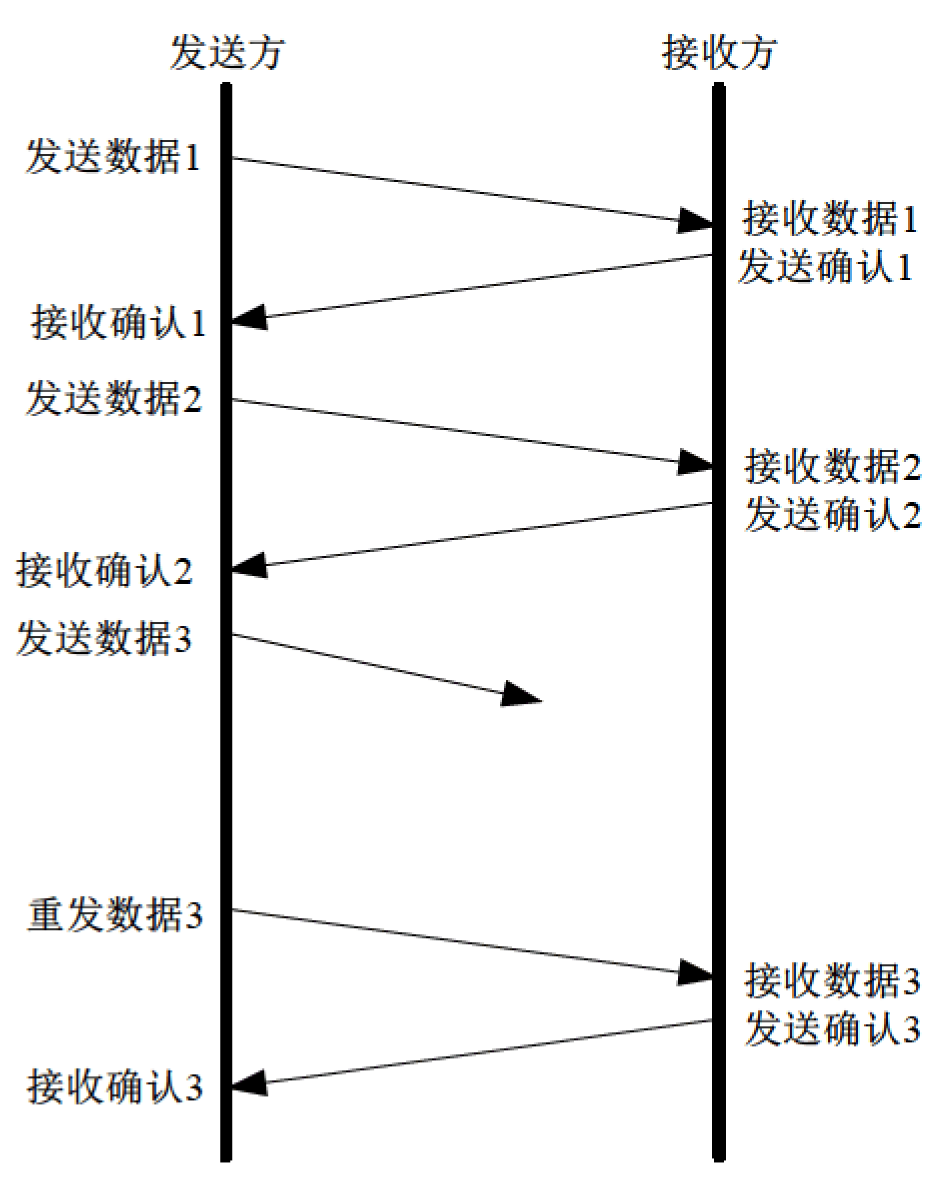


图 9.1　确认重发机制示意图

从图 9.1 中可以看出，发送方发出一个数据后总是要等接收到接收方发出的对该数据的确认后才会发送下一个数据。如果发出数据后等待一定的时间还没有接收到接收方的确认（如图中第一次发送数据 3），发送方就认为该数据已丢失，并重新发送该数据。通过这种机制确实能够提供可靠的传输服务，除非发送者和接收方之间的网络连接已经断开。这种机制保证了丢失的数据包可以通过再次发送增加其到达目标的可能性。但在这种机制中还有一个问题。因为发送方发送数据后过了一定的时间没有收到确认就会重发数据。但有时候数据包不是真的丢失了，而是网络传输的延时太长，以至于在很久之后才到达目标主机。这样的话目标主机就会收到 2 次相同的数据，而发送方也会收到同一数据的两次确认。那么怎样区分收到的数据和确认是否是同一个呢？通常的做法就是在数据包中记录一个数据的序列号，相同的数据如果发送多次都使用同一个序列号。接收方发送确认数据包时也要注明是对哪一个序列号的数据包的确认。就像图 9.1 中的 1、2 和 3 就分别表示不同数据的序列号。

TCP 协议就是通过超时重发机制来提供可靠服务的。但从图 9.1 中可以看出，这种机制的效率是很低的，因为在双方等待的过程中，网络非常空闲。TCP 使用一种称为滑动窗口的机制可以提高网络的利用率，从而提供数据传输的效率同时又能保证可靠性，我们将在后面讲述这种机制。

9.1.3　面向字节流的传送服务

不管 TCP 传输数据的机制是什么，它向上层应用程序提供的传输服务是面向字节流的。TCP 协议将应用程序需要传输的数据理解为按顺序排列的二进制位，并按 8 位字分隔，形成有序的 8 位字流，也称为字节流。发送方将数据以字节流的顺序递交给下层的 TCP 协议进行传输，接收方的 TCP 协议会将数据以相同的字节流顺序交给接收方的程序。虽然 TCP 协议数据的传输最终是以数据包的形式进行的，但 TCP 协议在组织数据包时并不会关心数据的结构，而是以它觉得合适的方式将数据进行分割。因此发送方的应用程序将一个完整结构的数据组织成字节流一次交给 TCP 协议传输后，接收方的应用程序通过一次收取动作可能无法获得完整的记录数据，因为 TCP 协议可能将该数据进行了分割，而目前到达的数据可能只是整个记录数据的一部分。因此接受方的应用程序在接收数据时必须自己识别记录的结构，如果发现一次收取的数据不是完整的记录就需要进行多次收取并将多次收集的数据还原成完整的记录。

9.2　TCP 协议数据段的格式

TCP 协议向应用程序提供的服务是面向字节流的，应用程序不用关心这些字节流是怎样进行传输的。但 TCP 数据的传输是通过 IP 协议进行的，IP 协议的传输单位是 IP 数据包。因为用户提供的字节流数据可能很大，而一个 IP 数据包所能容纳的数据是有限的（至少在理论上受到 IP 数据包头部中包长字段最大值的限制），因此 TCP 协议必须将字节流数据进行分割并组织成 IP 数据包进行传输，在目标主机的 TCP 协议将这些分割的数据再组织成数据流。TCP 协议数据包有自己的头部和数据区，一个 TCP 数据包成为段。本节来看看 TCP 数据段的具体格式，并对其中部分字段的意义进行说明，而其他一些字段的意义将在后面讲到 TCP 协议的各种机制时进行解释。

9.2.1　TCP 数据段的格式

图 9.2 就是 TCP 数据段的具体格式。

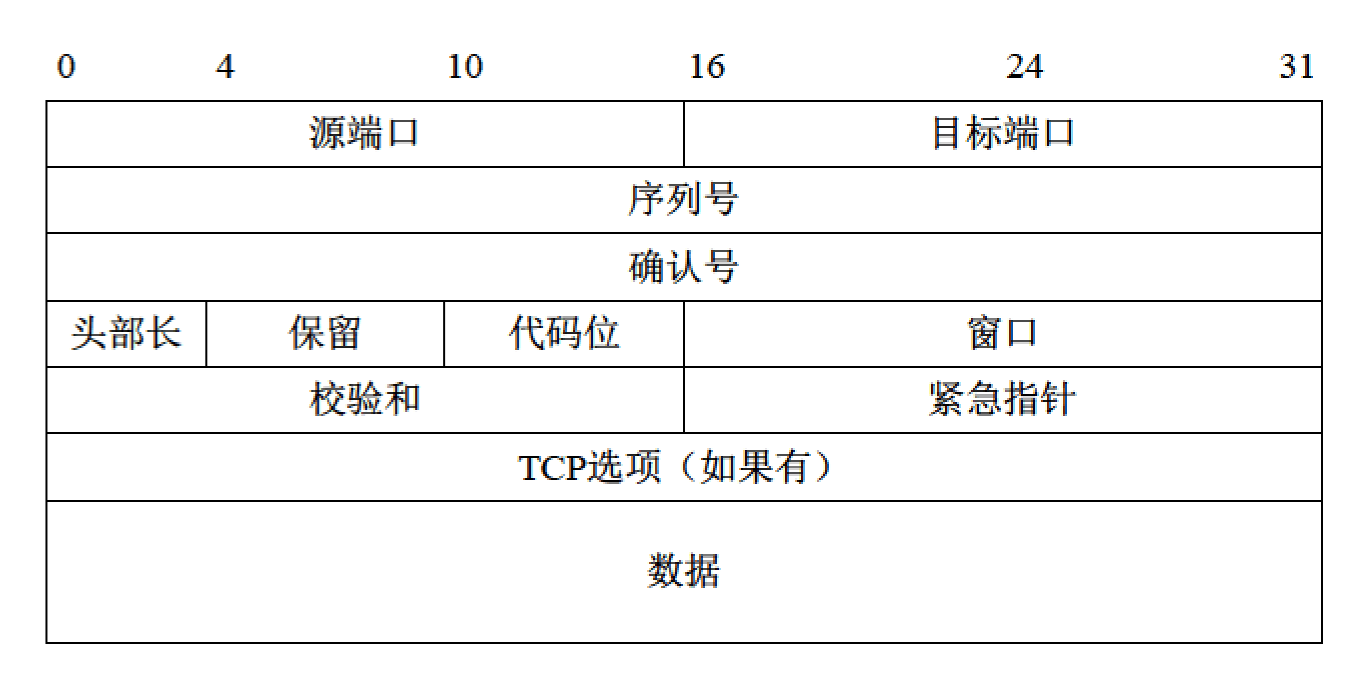


图 9.2　TCP 数据段的格式

源端口和目标端口用于指定发送方和接受方的 TCP 端口号。和 UDP 协议不同的是，TCP 段中源端口号必须指定。这是因为我们提到过 TCP 是面向连接的，一个 TCP 连接由发送方和接收方的 IP 地址和 TCP 端口号组成。因为一个 TCP 端口号可以为不同的连接所重用，所以两个端口号中缺少任何一个都无法确定该数据段所属的 TCP 连接，也就无法确定处理数据的应用程序了。

头部长字段的值是 32 位计的 TCP 段头部的长度。因为 TCP 头部有一个选项字段是可选的，所以需要这个字段来区分 TCP 头部和数据区。

另外，头部有一个代码位字段，该字段的长度是 6 位，这 6 位从前往后分别称为 URG、ACK、PSH、RST、SYN 和 FIN。这 6 位的设置代表了对段头部其他字段意义的解释。在后面讲到 TCP 协议的各种相关机制时再一一说明这些位的作用。由于序列号和确认号字段要到后面才解释， TCP 连接的建立要使用重发与确认机制，这就涉及到序列号的初始化和确认问题。在讲到 TCP 连接的建立时会忽略这些问题，直到讲到 TCP 数据传输机制时再详细解释。

9.2.2 TCP 校验和的计算

TCP 数据段头部校验和字段的长度为 16 位，它的作用是用来保证传输过程中 TCP 数据段的完整性。校验和在发送方计算好并填入校验和字段。接受方以同样的方法计算数据段的校验和并与发送方计算的进行比较，如果一致就说明数据段在发送过程中没有被改动，否则就被改动了。只有发送方和接收方计算的校验和一致的数据段才会被接受。

和 UDP 使用 UDP 伪头部计算数据包校验和一样，TCP 校验和的计算不仅包含了 TCP 数据段中的所有数据还包括一个称为伪头部的结构和将 TCP 数据段补足 16 位的整数倍的一个全为 0 的 8 位字。计算校验和时，TCP 协议先构造该数据段的一个伪头部结构，然后将 TCP 数据段的校验和字段设置为 0 并将其连接在伪头部后面，最后将 TCP 数据段的长度补足为 16 位的整数倍。最后按照 IP 协议校验和的计算方法对这个新的结构计算校验和并将结果填入校验和字段。TCP 伪头部和长度补足部分不会进行传输。

TCP 伪头部的格式如图 9.3 所示。

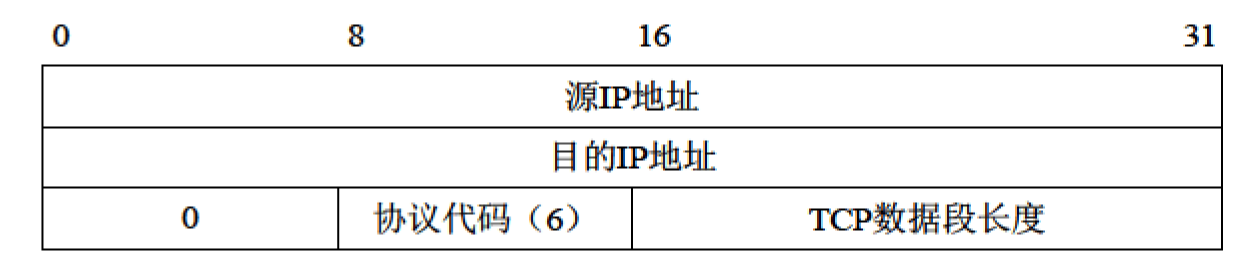


图 9.3 TCP 伪头部结构格式

源 IP 地址和目的 IP 地址字段包含了发送该数据段的源主机和接收它的目的主机的 IP 地址。协议代码字段为 TCP 协议的代码。TCP 数据段长度字段就是 TCP 数据段长度，不包括伪头部和补足部分的长度。

使用 TCP 伪头部的目的是为了让数据段的接收者确定发送和接收的 TCP 数据段是来自正确的源地址且该数据段是发给自己的。我们知道在 TCP 数据段的结构中只包含了数据段的源端口和目的端口，而没有包含源 IP 地址和目的 IP 地址。因此，TCP 使用伪头部结构来计算校验和。在发送方构造一个伪头部结构与待发送的 TCP 数据段一起计算校验和后发送给接

收方。在接收方使用同样的方法计算校验和并与发送方计算的校验和进行比较，如果两个校验和匹配，就说明该数据包是发给本主机的，且数据传输没有出错。

9.3　TCP 协议连接的建立与关闭

TCP 协议是面向连接的传输协议，一个 TCP 连接由发送方的 IP 地址与 TCP 端口号和接受方的 IP 地址与 TCP 端口号标识。建立一个 TCP 连接的作用就是让发送方和接收方都做好准备，准备好之后就要开始进行数据传输了。在本节中，将解释 TCP 连接的建立与关闭过程。

9.3.1　被动打开与主动打开

在第 8 章中讲到 UDP 端口时讲到第一个发送 UDP 数据包的应用程序必须知道接受方的 UDP 端口号，且接收方的应用程序必须已经在等待该 UDP 端口上的数据。这样，使用 UDP 的通信才能进行。这说明一个使用 UDP 进行通信的最低要求，即必须存在一个主动方和一个被动方。被动方的端口必须是主动方和被动方已经约定的，主动方向被动方发送第一个数据并告诉被动方自己的端口。

建立 TCP 连接也一样。因为 TCP 连接的建立过程也是通过连接的两方之间进行消息传输进行，这就要必定有第一个消息（连接请求）的发送。发送连接请求的是主动方，等待连接请求的就是被动方。主动方发出连接请求后，只有被动方已做好接收该请求的准备后面的过程才能够继续进行。

应用程序向操作系统申请一个 TCP 端口并做好等待其他主机的连接请求的过程称为 TCP 连接的被动打开。相反，应用程序向操作系统申请一个 TCP 端口并主动向其他主机发出连接请求的过程称为主动打开。被动打开的连接只有在接收到其他主机的连接请求后才会产生一个新的连接，而如果还有其他主机向该被动打开的连接发出连接请求又会产生新的连接。

9.3.2　三次握手建立 TCP 连接

所谓的三次握手就是要有三次连接信息的发送/接收过程。TCP 连接的建立需要进行三次连接信息的发送/接收，如图 9.4 所示。

三次握手的目的主要在于同步连接双方发送数据的初始序列号。首先，连接的主动打开方（即连接请求方）向被动打开方（即连接接受方）发送一个 TCP 数据段，该 TCP 段通常不包含数据区，并将代码位中 SYN 位置 1 设在序列号字段设置一个初始序列号。被动打开方接收到这个连接请求段后，向主动打开方发送一个 TCP 段。将 ACK 位置 1，表示已经收到了主动打开方的连接请求。被动打开方还会将 SYN 位置 1 并在序列号字段设置自己的初始序列号。主动打开方接收到被动打开方的 TCP 段之后会发送一个 ACK 给被动打开方，接收方接收到该数据段后连接的建立就完成了。现在可以开始传输数据了。

虽然上述过程中是一个主动方向被动方发起连接，但 TCP 连接也有可能是在两个主动打开方之间建立的。图 9.5 就说明了该过程。

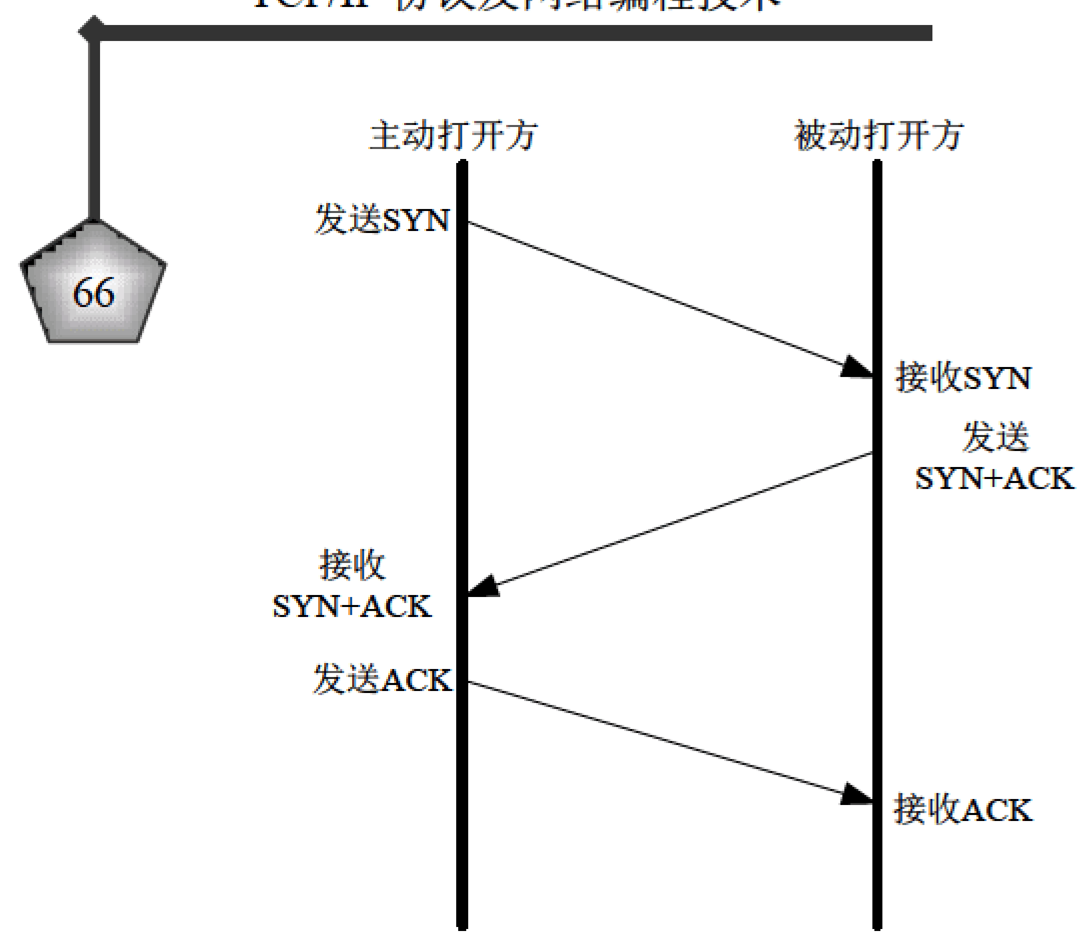



图 9.4　TCP 建立连接的三次握手过程

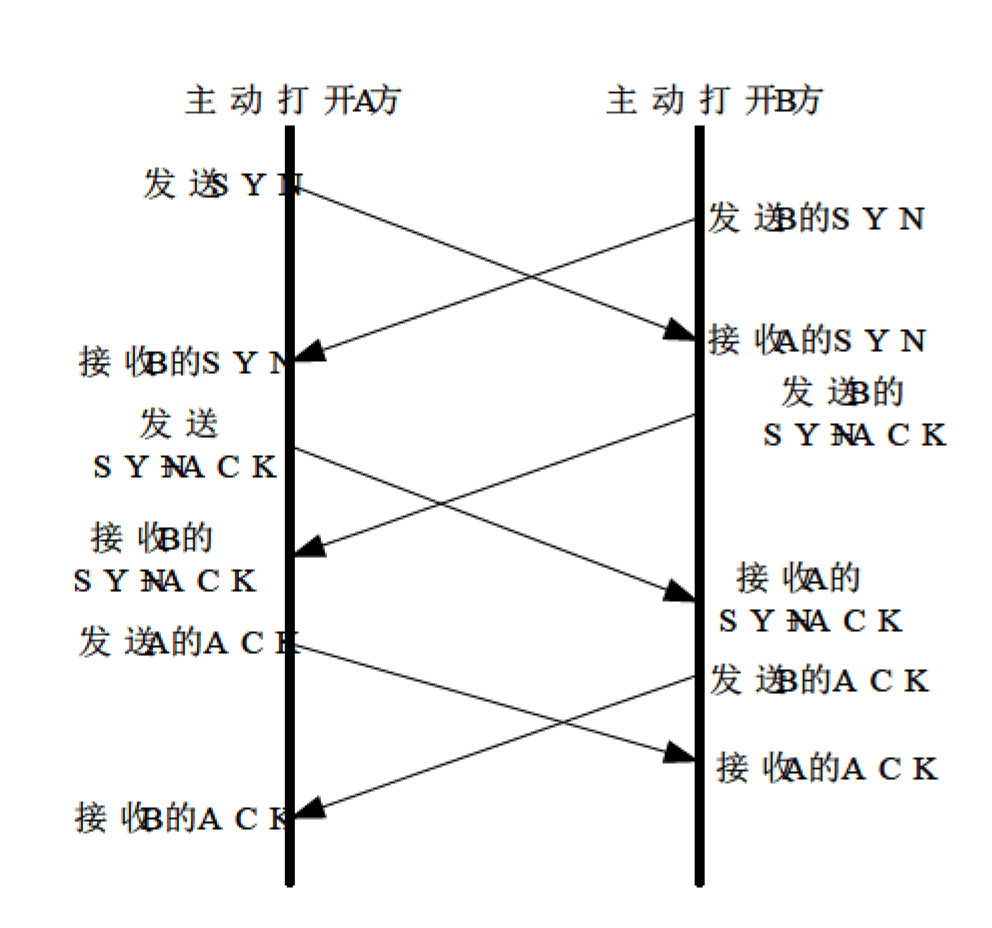


图 9.5　两个主动打开连接建立的过程

需要注意的是，图 9.5 中 A 和 B 发送的 SYN 和 SYN+ACK 段中的序列号字段应该是相同的，否则接收方就不知道它的初始序列号是什么了。

最后要提到的是，一般建立连接的 TCP 段中是不携带数据的，即数据区为空。但这并不说明这些段中不能携带数据。如果上述的各种 TCP 段中携带了数据，段的接收方 TCP 就必须先将这些数据保存下来，等连接建立后就可以迅速递交给上层应用程序处理了。

9.3.3　TCP 连接的关闭

TCP 连接建立之后就可以开始传输数据了，在所有需要传输的数据传输完之后，就需要将 TCP 连接关闭，以释放为连接使用的资源。

TCP 连接是一种全双工的连接，即一个 TCP 连接的两个端点之间可以同时发送和接收数据，而不是每一个时刻只能有一个端点发送数据。可以假设每个 TCP 连接中都有两个管道，它们的传输方向是相反的。这样就能实现双向的数据传输。因此 TCP 连接的关闭就出现了一个半关闭的概念。所谓半关闭的意思就是只关闭一个方向的数据传输，另一个方向的数据传输还是可以继续的，即只关闭了其中的一个方向的管道。

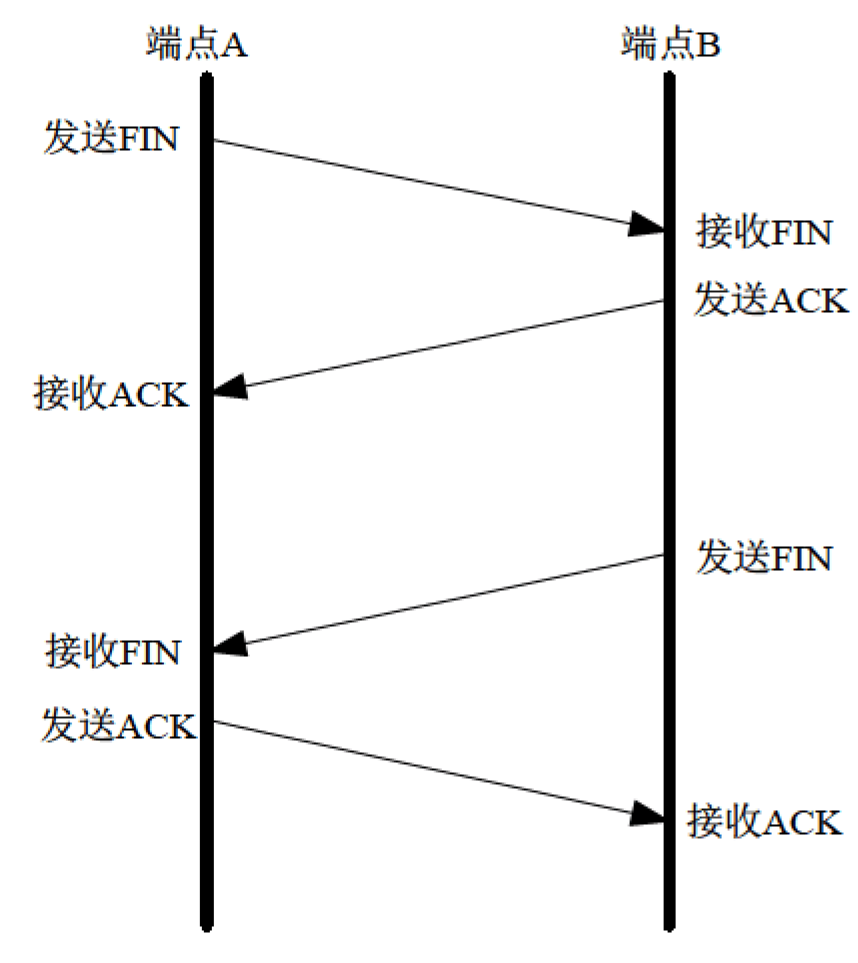


图 9.6　TCP 连接的关闭

这样关闭一个 TCP 连接就需要 4 个步骤，如图 9.6 所示。

在图 9.6 中，端点 A 先关闭了连接。这样，从 A 发往 B 的数据就完成了。A 通过向 B 发送一个 TCP 段并将段中的 FIN 位置 1 来关闭连接中从 A 到 B 的管道。在 B 收到该数据段之后，B 会发送一个 ACK 段给 A。但 B 不一定会立刻关闭从 B 到 A 的管道。因为 B 可能还有

数据需要发送给 A。只有在 B 的数据传输完后 B 的应用程序才会关闭该连接。这时 B 通过将发往 A 的数据段中的 FIN 置 1 来通知 A 所有数据已发送完毕。A 发送一个 ACK 段给 B。这样整个 TCP 连接就关闭了。

上面的连接关闭方式是在所有数据传输完后正常关闭连接的方式，但并不是所有的时候都能向上述那样“礼貌”地关闭连接。通常在发生异常时，一方可能需要立即关闭 TCP 连接，而无法等待对方的数据传输完成。这种情况下，要关闭连接的一方向对方发送一个 RST 位置为 1 的 TCP 段，并不再等待该段的 ACK，直接将连接关闭。在另一方，收到该数据段后也直接将连接关闭。与上面的关闭连接的方式相比，这种方式要“野蛮”得多。这种连接的关闭方式通常称为复位。

9.3.4　TCP 连接状态迁移

因为 TCP 协议的连接从建立到关闭的整个过程中，在不同的时候收到数据段会有不同的处理，可以用连接的状态来表示这些不同的时间，并使用状态迁移图来说明 TCP 在各种状态下对各种数据段的处理方式。这样有助于理解 TCP 连接的建立与关闭过程。图 9.7 就是这样一个 TCP 连接的状态迁移图。

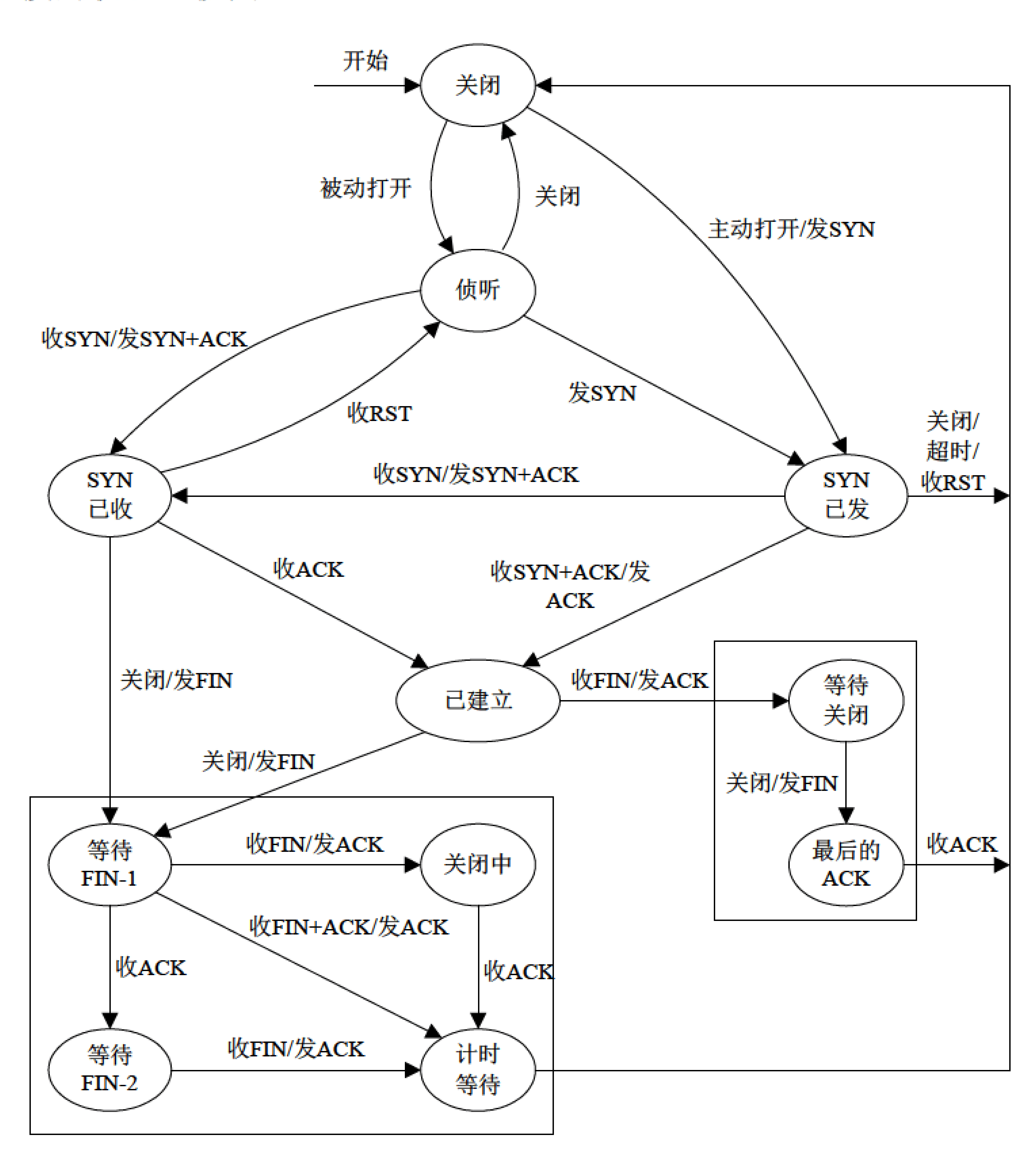


图 9.7　TCP 连接状态迁移图

在图 9.7 中，圆表示连接状态，箭头表示状态的迁移。箭头线旁的标签标识发生该迁移时收到的数据段及对该数据段作出的反应。

从图 9.7 中可以看出，一个 TCP 连接可以有 11 种不同的状态。其中左下角的四个状态是应用程序主动关闭（即先发送 FIN 数据段）经过的状态，右边虚框中的两个状态是被动关闭经过的状态。

计时等待状态也称为 2MSL 等待状态。每个具体 TCP 实现必须选择一个 TCP 段最大生存时间 MSL（Maximum Segment Lifetime）。它是任何 TCP 段被丢弃前在网络内的最长时间。这个时间是有限的，因为 TCP 段以 IP 数据包在网络内传输，而 IP 数据包则有限制其生存时间的 TTL 字段。MSL 通常为 2min。对 MSL 处理的原则是：当 TCP 执行一个主动关闭，并发回最后一个 ACK，该连接必须在计时等待状态停留的时间为 2 倍的 MSL。这样可让 TCP 再次发送最后的 ACK 以防止这个 ACK 丢失（另一端超时并重发最后的 FIN）。这种 2MSL 等待的另一个结果是这个 TCP 连接在 2MSL 等待期间，定义这个连接的端口不能再被使用。

9.4　TCP 协议数据的传送与流量控制

本节具体介绍 TCP 协议的数据传输与流量机制，将涉及到字节流的分段、TCP 的确认重发机制、超时的判断及紧急数据的传输等。

9.4.1　字节流的分段

因为 TCP 向应用程序提供的是面向字节流的传输服务，所以应用程序可以随意地向应用程序提交字节流数据。极端的情况是应用程序每次仅提交一个字节的数据让 TCP 协议进行传输。如果 TCP 协议每次收到应用程序提交的数据都组织一个或多个数据段进行传输，而不管数据的多少，那么在用户每次都提交少量数据而提交的次数很多时数据的传输效率是很低的。

为此 TCP 协议采用缓冲的办法，这就类似于操作系统的写文件操作。当应用程序打开一个文件进行写操作时，操作系统就为该文件提供了一个缓冲区。应用程序写往该文件的数据都先存放在该缓冲区，并没有真正的写到文件中。只有在缓冲区已满或者应用程序显式地要求将数据写入文件时，缓冲区中的数据才会写入磁盘文件中。TCP 使用的是同样的方法。应用程序提交给 TCP 协议的数据会存放在一个缓冲区中，并没有真正地发送到连接的另一端。只有在合适的时候或者应用程序显式地要求将数据发送出去时 TCP 才会将数据组织成合适的数据段发送出去。至于 TCP 要等到缓冲区聚集了多少数据才开始发送，一般要根据 TCP 协议确定的最大段长度决定。

我们知道 TCP 数据段是封装在 IP 数据包中进行传输的，从理论上讲，TCP 数据段的最大长度加上 IP 数据包头的长度不能超过 65 535 个 8 位字。但实际上的 TCP 数据段的最大长度要远远少于这个值，因为它要受到许多其他因素的影响。

首先，我们在第 3 章中提到 IP 数据包的分片问题。因为 IP 协议的数据包是封装在底层的物理网络数据帧中进行传输的，而很多物理网络都对数据帧的最大长度有限制，且通常都比 IP 数据包的理论最大长度小的多。所以，如果一个 IP 数据包太大就需要将其分成多个较

小的能够封装在一个数据帧中的小数据包中进行传输。这样做是不得已的，因为这会引起几个问题：第一，因为大的数据包要分解成小的数据包，在目的地又要重新组合成大的数据包，且在传输过程中需要对每个数据包进行路由。所有这些都影响了数据传输的效率。第二，因为分成多个分片的数据包必须在目的地进行重组，所以这些分片一个都不能少，必须全部到达目的地，如果有一个丢失那么所有的分片都要重新进行传输。因此分片增加了传输失败的几率。所以在进行数据传输时应该将 TCP 数据段的大小控制在合适的范围，使得它能够在最终封装在 IP 数据包中之后能通过一个数据帧进行传输。

另外还有一个因素限制了 TCP 数据段的最大长度。因为网络上的主机系统的计算能力和资源的差别可能是很大的。大型服务器的内存数量可能是几个 GB，而一些嵌入式系统的内存可能只有几个 KB。如果能力相差如此悬殊的系统之间进行通信，就必须让能力强的一方知道能力弱方所能接受的包的最大长度，以免一个过大的数据包将该主机系统淹没。

从以上分析中可以看出，TCP 数据段最优的长度就是在使得 IP 数据包无需分片且接收方能够接收的情况下，尽可能在一个段中多传输数据。为了让发送方知道接收方的接收能力，TCP 使用了一个选项。在建立连接时，连接的任何一方都可以在建立连接的段中包含一个最大数据段长选项，用以通知对方自己能够接收的最大数据包的长度。选项的格式如图 9.8 所示。

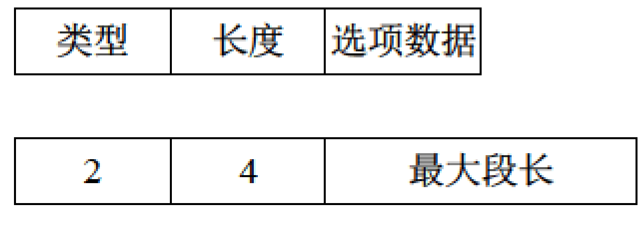


图 9.8　TCP 选项格式

TCP 选项通常由 3 部分组成：类型、长度和选项数据，但有的只有类型字段。类型字段的长度是 8 位，长度字段的长度也是 8 位。长度字段的值表示整个选项部分的长度，而不是只选项数据字段的长度。最大段长选项的类型代码为 2，选项数据字段的长度为 16 位，所以整个选项的长度为 4。

虽然要确定接收方的接收能力很容易，但要确定从发送方到接收方之间的网络的最小 MTU 却是一件很困难的事情。首先，由于 TCP 处于 IP 层之上，它无法知道底层网络的 MTU。其次，IP 数据包的传输路径是动态的，即同一个 TCP 连接的数据段封装成的数据包可能会通过不同的路径到达接收方。因此，发送方的 TCP 无法得知要发送的数据段会从什么路径进行传输，也就无法确定该路径上的 MTU 了。

9.4.2　滑动窗口机制

在 9.1.2 节提到 TCP 协议是通过确认重发机制来在不可靠的 IP 协议的基础上提供可靠的传输服务的。但当时所介绍的确认重发机制是一种低效率的机制，因为 TCP 在等待数据或确认时网络是很空闲的。实际上 TCP 使用的确认重发机制是一种称为滑动窗口的机制。在本节中先简单说明基本的滑动窗口机制的原理，然后再具体说明 TCP 协议中使用的滑动窗口机制。

9.4.2.1　基本滑动窗口机制

简单的确认重发机制之所以效率低下是因为发送方在发出一个数据后就必须停下来等待对方的确认，也就是说在这段时间里从发送到接收方的路径上只有一个数据在传输，即只有一个设备是忙的，其他设备都是空闲的。滑动窗口机制的改进在于它让发送方可以连续发送多个数据，然后等待接收方的确认。这样就大大提高了网络的利用率。现在开始具体解释

滑动窗口机制的工作原理。

可以将 TCP 协议发送的字节流根据状态分为三个部分：第一部分是已经发送过且收到了对方确认的，TCP 能确定这些数据已被对方收到。第二部分为已经发送但还没有收到对方确认的，这些数据可能正在传输途中，也可能对方已经收到但正在传输这些数据的确认，还有可能数据确认已经丢失了。第三部分的数据就是 TCP 还没有发送的数据。图 9.9 就表示了这种划分关系。

已发送	发送中	待发送

图 9.9　TCP 数据状态划分

对于已发送的数据，TCP 协议可以完全不用关心，也不用保存。对于待发送的数据，可能应用程序还没有提交给 TCP 协议，也有可能以字节流的形式保存在 TCP 协议的缓冲区中。而对于发送中的数据，TCP 协议已经构造好了数据段，而且 TCP 协议必须为这些数据段启动一个定时器。在定时器超时后如果没有收到确认就必须将数据段进行重发。发送中的数据可以是多个数据段，但必须是有限的，要根据连接双方的资源量而定。初始情况下已发送部分的长度是 0，当不断发送数据并不断收到确认时，图中的灰色区就不断地向右移动，直到所有数据发送完毕。这个灰色区域就好像是整个数据中的一个窗口，且该窗口是不断移动的，所以称为移动窗口机制。

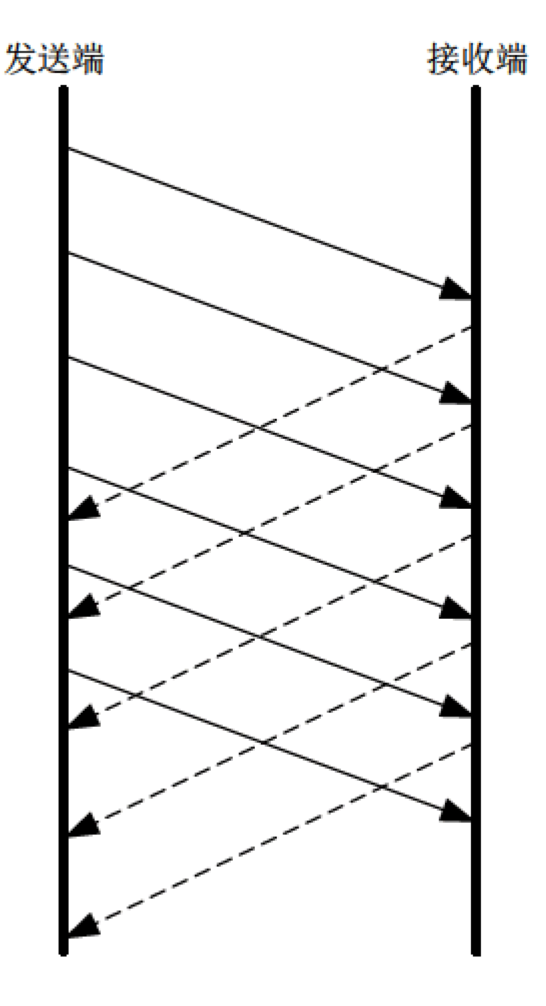


图 9.10　移动窗口机制对网络资源的利用率

移动窗口机制中窗口的大小决定了数据传输的效率。在极端的情况下窗口的大小为 1，那就是前面讲过的简单确认重发机制。如果窗口的大小合适，就可以最大限度地利用整个网络资源。图 9.10 就说明了这个原理。

我们假设图中是一种理想状态，即数据发送有延时但不丢失。这样可以简化要说明的问题但不会影响它的本质。实线表示发送的数据，虚线表示对接收到的数据的确认。实线之间的间隔表示发送方准备数据需要的时间。那么在图中发送方把窗口的大小设为 4 就可以最佳地利用网络资源了。首先，发送方可以一次发送 4 个数据而不用等待对方的确认。它刚发送完第 4 个数据就收到了第 1 个数据的确认，这样它就可以将窗口向右移动一个位置，这时就可以发送第 5 个数据了。依此类推。从图中可以看出，这种机制极大地利用了 TCP 这种全双工连接的特点同时进行双向传输。图 9.11 就说明了图 9.10 中发送窗口的移动情况。

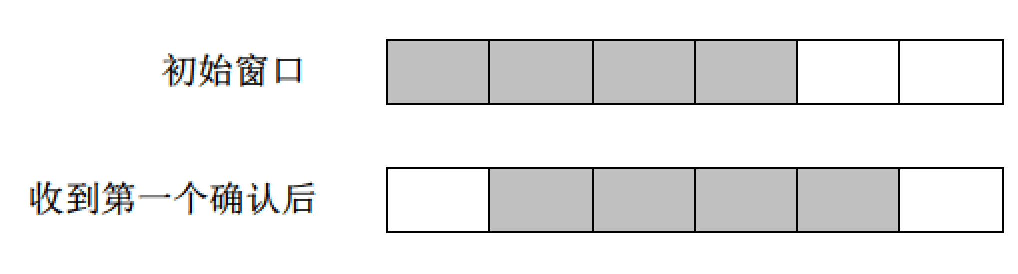


图 9.11　窗口移动过程

9.4.2.2　TCP 数据段的标识－序列号

因为数据段在使用 IP 数据包进行传输时有可能会丢失、失序或重复。所以发送方和接收方都要有一种机制来区分收到的数据段或确认是否是同一个，并区分数据段的先后关系。TCP 数据段头部有一个序列号字段，该字段就是用来标识不同的数据段的。后面的数据段序列号大于前面数据段的序列号。但 TCP 协议中的序列号不是数据段的编号，而是按字节进行编号的。例如前一个数据段的序列号是 216 且该数据段的数据区携带了 100B，那么后一个数据段的序列号应该是 316。

另外，TCP 数据段头部还有一个确认号字段，该字段作用是用来向数据发送方确认它已经收到的数据段。TCP 协议之所以需要序列号和确认号两个字段是因为它允许连接的两方在发送数据时对对方的数据段进行确认，而不是使用单独的确认段进行确认。因为 TCP 是全双工的连接，连接的两端可以同时向对方发送数据。因此这种确认方式减少了数据段的数量，提高了数据传输的效率。需要指出的是，TCP 对数据段的确认号不是简单的确认收到的数据段的序列号，而是表示它准备接收的下一个数据段的序列号。例如 TCP 收到一个数据段的序列号是 216 且该数据段的数据区携带了 100B，那么它发送的数据段中应该将确认号设置为 316，表示它希望接收序列号为 316 以前的数据都已经收到了。还需要指出的是，确认号字段不是在每个数据段中都用到的，只有数据段中设置了 ACK 位才表示确认号字段有意义。

虽然字节流的第一个字节的编号总应该是 1，但 TCP 中携带字节流中第一个字节的数据段的序列号并不总是 1，这是因为 TCP 协议规定每个 TCP 连接的初始序列号必须是随机产生的一个值，这个值是在建立连接的过程中指定的。我们回顾一下 TCP 连接的建立过程。当时没有提及序列号的问题，现在重新来解释这个问题。

连接发起方的一端的数据段中必须将 SYN 位置 1，同时它还需要产生一个随机的初始序列号并将数据段的序列号字段的值置为该值。被动打开方在发送 SYN+ACK 之前也必须产生一个自己的随机初始序列号。整个连接建立的完整过程如图 9.12 所示。

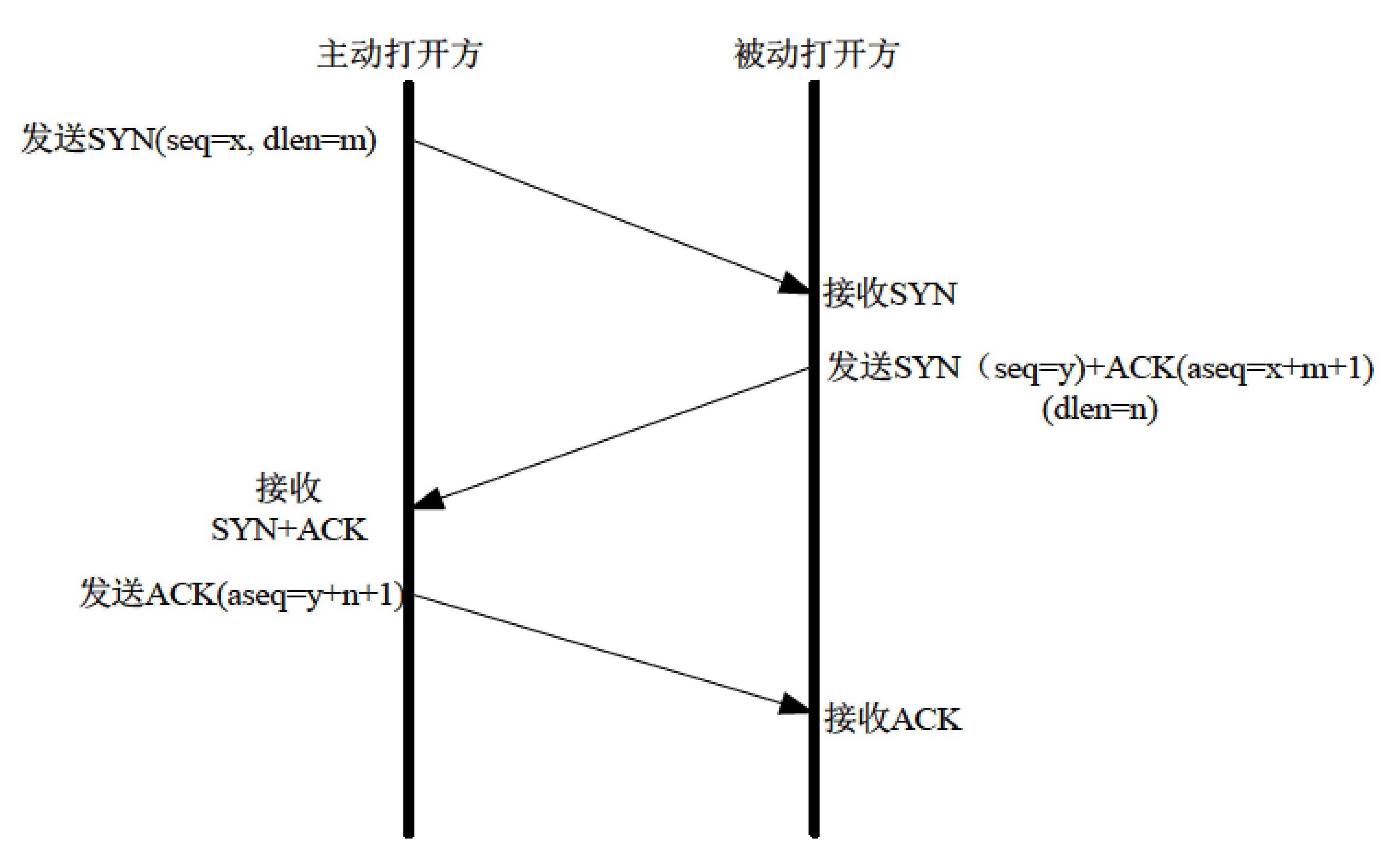


图 9.12　TCP 连接过程中初始序列号的确定

图 9.12 中，seq 表示序列号字段，aseq 表示确认号字段，dlen 不代表任何字段，而是指

数据区的长度。虽然通常建立连接的数据段中都不携带数据，但并不是不可以携带。

9.4.2.3 TCP 的滑动窗口机制

前面是以数据包为单位说明基本滑动窗口机制的工作原理。在 TCP 协议的滑动窗口机制中，基本的数据单位不是数据段，而是字节。在 TCP 协议中，每个窗口都用 3 个指针表示，如图 9.13 所示。

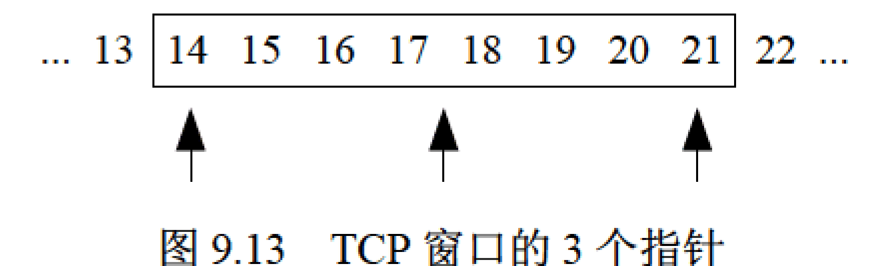


图 9.13　TCP 窗口的 3 个指针

第一个指向窗口的第一个字节，第二个指向窗口中马上要发送的字节，第三个指向窗口的最后一个字节。除了发送方需要一个窗口外，接收方也要有一个窗口表示当前需要接收的数据，但接收方的窗口无法用 3 个指针表示，原因如图 9.14 所示。

... 200 | 201 500 | 501 732 | 733 800 | 801 1000 | 1001 ...

图 9.14　TCP 数据接收方窗口状态

图中每个实线框表示接收方的当前窗口，灰色部分表示已收到的数据。因为数据段的失序使得接收方无法将这些数据还原为字节流，所以它必须先暂存这些数据。但接收方也无法就已收到的部分数据向发送方发出确认，因为 TCP 数据段的确认号字段表示的是接收方希望接收的数据的编号，即该编号之前的数据都已接收到了。如果下一次一个序列号为 201，数据区长度为 300 的数据段到达，则接收方可以将下一个发送的数据段的 ACK 位置 1 并将确认号字段设为 733 了。这样接收方和发送方的窗口就可以向右移动到 733 位置了。

因为 TCP 连接是全双工的，所以连接的每一端都必须保存两个窗口，一个用于发送数据，一个用于接收数据。

TCP 的这种确认方式称为累积确认。这种确认方式有它的优点和缺点。

先看累积确认的优点。首先，它允许发送方在对数据进行重发时发送更多的数据。举例说明这个问题。例如发送方的应用程序先递交了 500B 的数据给 TCP 传输，等 TCP 将该数据发出后应用程序又递交了 300B。这时如果 TCP 没有收到签名 500B 的数据的确认，它就可以将 800B 数据一次发送出去。这就增加了前面 500B 数据传输成功的几率。其次，累积确认无需对丢失的确认进行重发。因为确认数据段也会丢失，但只要下一个确认能够到达对方就无需对前一个确认进行重发。

为了说明累积确认的缺点，考虑图 9.15 所示的情况。

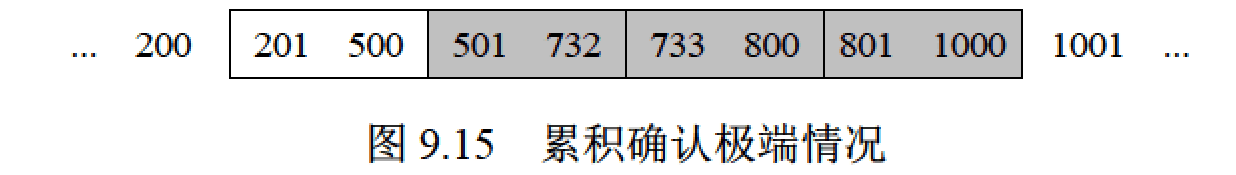


图 9.15　累积确认极端情况

图 9.15 是接收方的窗口，每个方框表示一个数据段。窗口的大小位 800B，发送方为 4 个数据段将一个窗口内的数据发送给接收方。在传输途中，第一个数据段丢失。发送方发送了数据后就启动一个计时器，等待对方确认。但由于接收方没有收到第一个数据段，所以它

发送的每个数据段的确认号都为 201。发送方超时后就要重发数据，但它是应该重发所有数据呢？还是应该只发送第一个数据段的数据？如果它重发所有的数据，那后面 3 个都是多余的。将它们进行重发不但浪费两方的计算资源，还会浪费网络带宽。虽然目前的标准做法是重发第一个数据段。但这也有一个问题，如果接收方确实一个数据段都没有接收到，那么 TCP 现在的机制就回到了最原始的简单确认重发机制，TCP 每次只能发送一个数据段。标准做法之所以只发送一个数据包主要是为了保护整个网络，因为所有 TCP 连接都重发所有数据包会极大地增加网络的负担，甚至造成网络崩溃。

现在看 TCP 协议中窗口的大小。显然，窗口是越大越好，数据传输的效率会越高。但从 TCP 的移动窗口机制可以看到维护一个窗口是需要资源的，至少需要和窗口大小相同的缓冲区。对于一些内存资源紧张的系统（如嵌入式设备）来说，其窗口不可能设置得很大。由于发送方会将其当前窗口内的数据一次性的发送出去，这就要求发送方的窗口不能大于接收方的端口，否则接收方可能无法容纳超过其窗口大小的数据。虽然接收方可以选择丢弃窗口之外的数据，不至于造成系统崩溃，但这就意味着该数据的发送是一种资源的浪费，根本不应该发送。所以，发送方应该根据接收方窗口的大小来设置自己的窗口大小，使其不大于对方的窗口。这就需要有一种机制来让接收方将自己的窗口大小通知给发送方，这是通过使用 TCP 数据段头部的窗口字段来完成的。在 TCP 协议中，凡是 ACK 置为 1 的数据段都应该将自己接收窗口的当前大小填入窗口字段中。这样做就允许接收方随时改变窗口的大小。但有一点需要说明的是，新窗口的右边界不能小于原窗口的右边界。例如接收方当前的窗口为 101～1000，那么在它下次收到一个序列号为 101 数据区长度为 100 的数据段之后，它可以在确认数据段中将窗口大小设为 800，但不能设为 799。虽然缩小窗口大小有限制，但接收方可以随时扩大窗口的大小。还是上例，即使接收方没有收到任何数据段，它也可以在确认数据段中将窗口大小改为 1000。缩小窗口的最极端的例子就是将窗口的大小设为 0，这样就停止了数据传输。

那么接收方为什么需要一个可变大小的窗口呢？这不但和接收方的存储能力有关，还要受到接收方的计算能力的限制。为说明这个问题，看图 9.16 所示的例子。

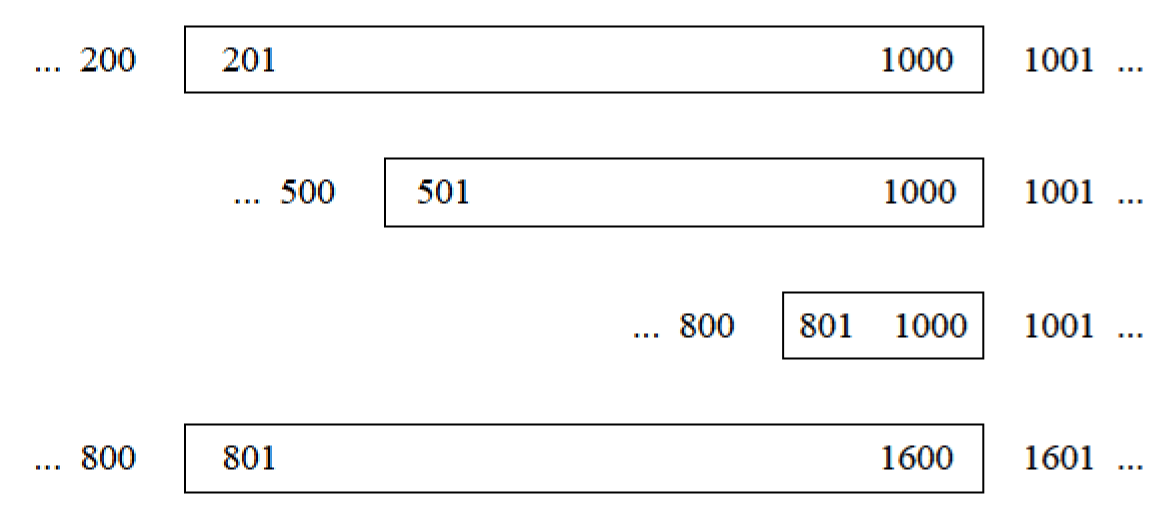


图 9.16　TCP 接收窗口大小变化示例

图 9.16 中，接收窗口的初始大小为 800B。图中最上面的图表示 TCP 协议在等待数据。第二个图表示收到了 201～500 的数据，所以窗口的左边界向右滑动到 501。但上层应用程序由于某种原因（如数据发送过快，来不及处理或某部分需要人工干预，正在等待等）无法接收数据，导致缓冲区中的数据无法提交。这无形中就减小了缓冲区的大小，使得窗口的右边界无法向右移动。接收方通过在 ACK 数据段中将窗口大小改为 500 通知发送方，发送方就



相应地减小（如果需要）发送窗口的大小。如果下次收到数据还无法提交给上层应用程序，窗口将继续减小。一旦应用程序处理完数据，TCP 协议可以将积压的所有数据提交上去，窗口就可以恢复原来的大小了。

综上所述，TCP 协议使用的是以累积确认为基础，字节为单位的可变大小的滑动窗口机制。这种机制不但实现了数据的可靠、高效的传输，还实现了端到端的流量控制。

9.4.3 超时的判断

TCP 协议使用的可变大小的滑动窗口机制实现了可靠的高效传输和流量控制，但这种机制还是以确认重发为基础的。确认重发机制要求发送方在数据发出后启动一个计时器，在计时器超时后就假设数据丢失并进行数据的重发。在本节中将介绍 TCP 协议的超时判断机制。超时判断机制对于 TCP 协议至关重要，因为它直接影响到 TCP 协议的效率。

9.4.3.1 TCP 协议判断超时的难度

判断超时最简单的办法就是指定一个固定的超时值，启动一个时间间隔为该值的定时器，定时器超时后就进行重发。这个固定值的指定可以有两种选择：第一种可以选择一个最大的值，所有网络的数据传输加上确认传输的时间，即数据传输一个来回的时间（也称为 RTT，Round Trip Time），都不超过该时间。这样在该时间过后就可以确定数据确实已经丢失了。但这种机制是很低效的，因为可能大多数网络的传输时间远远低于这个值，多等待的时间是浪费，降低了传输的效率。第二种是选择一个较小的值，这样好像可以提高传输效率，但实际上并不会。虽然丢失的数据不需要等待多久就进行重发，但因此也造成了很多稍有延迟的数据包的重发。这样就使得发送方和接收方必须处理很多重复的数据，更严重的是造成了网络上有过多的重复数据包在传输。所有的主机都这样做就会增加网络负担，并降低网络传输效率甚至造成网络瘫痪。显然 TCP 协议不能使用这种方法。

首先来看看 TCP 协议的目标。TCP 协议提供的是可靠的服务，但这种服务首先应该是高效的，否则就没有应用程序会使用了。因此 TCP 协议应该尽量不重发已经传输成功的数据，而对于已经丢失的数据应该尽量早地重发。要达到该目的的前提是准确地估计网络的 RTT。但 TCP 要估计网络的 RTT 是一件很困难的事情，这是由 Internet 的复杂性决定的。

首先，在 Internet 中，一个数据包的发送可能是两个主机之间直接进行的，也有可能要经过很多的网络节点（交换机、路由器等）。不同的网络环境传输数据所需要的时间也是不同的，那么 RTT 也会不同。而且各种网络 RTT 间的差别是很大的。其次，即使是一个网络之中，数据的传输效率也会根据网络数据流量的情况发生变化。这在跨网络的通信中尤其明显。

对 RTT 的估计是判断超时的关键，虽然要估计网络的 RTT 是很困难的，TCP 协议还是有一套很灵活的机制来合理的估计 RTT。

9.4.3.2 RTT 的计算

首先，TCP 协议 RTT 的计算是面向连接的，即它为不同的连接分别计算 RTT。这是因为连接是由两个端点决定的，有的连接可能是一个物理网络中的两台主机甚至是一台主机内部，而有的连接可能需要跨越多个网络。它们的 RTT 显然是不同的。

TCP 采用取样统计的方式计算连接的 RTT。TCP 协议记录每个数据段发送的时间，在收到对该数据段中数据的确认后，计算两者之间的时间差，即新的 RTT 样本。然后使用下面的公式计算新的 RTT：

$$\text{RTT} = (a \times \text{旧的 RTT}) + [(1-a) \times \text{新的 RTT 样本}]$$

式中，a 是一个大于 0 小于 1 的数。从公式中可以看出，a 越大，新的 RTT 样本对 RTT 的影响越小，RTT 的值就越稳定；a 越小，新的 RTT 样本对 RTT 的影响就越大，RTT 的值就越不稳定。通常 a 都设置为 0.9。

在发送数据段后启动该数据段的定时器，TCP 协议要根据当前的 RTT 计算该定时器的超时值，计算公式如下：

$$\text{Timeout} = b \times \text{RTT}$$

b 是一个大于 1 的数，对 b 的值也有一个选择问题。如果 b 选择过大，对于已经丢失的数据就要等很久才会进行重发；如果将 b 设为 1，就显得太急了。数据传输稍微有点延迟就进行重发。标准建议将 b 的值设为 2。

9.4.3.3　Karn 算法

上面的方法很简单，好像前面提到的判断超时很困难是在危言耸听。其实不然，关于超时判断的机制现在还是刚开始。

判断 RTT 的原理很简单，就是不停地取样然后与原有的 RTT 一起计算新的 RTT。但应该清楚 TCP 在超时后会对数据进行重发。现在就假设一个 TCP 连接中一方发送了一个数据段之后启动定时器，并在定时器超时后重发了该数据段。过了一段时间收到了对该数据段中数据的确认。现在出现一个问题，TCP 在收到该确认段之后，应该把它作为最初发送的数据段的确认还是应该把它作为重发的数据段的确认呢？答案是选择任何一个都是错误的。

首先，如果将该确认作为最初发送的数据段的确认，那么取出的 RTT 样本肯定大于现在的 RTT。这样算出来的新的 RTT 也就大于现在的 RTT 了。如果 TCP 连接是跨多个网络的，数据的丢失率可能会很高。这就会造成连接的 RTT 会不断上升。在极端的情况下，如果每个数据段都至少丢失一次的话，RTT 会变得无穷大。因为每丢失一个数据段并在重发后收到确认时所取得的 RTT 样本都会大于当前的 RTT 值，所以会造成 RTT 值的不断上升。

那么，选择将重发后收到的确认作为重发数据段的确认呢？我们考虑在网络传输延时突然增大的情况下会发生什么？这时发送的数据并没有真正丢失，接收方在接到数据后就发出确认。但在该确认到达数据的发送方之前发送方的定时器就超时了。这时发送方会重发数据，但过不了多久就收到了确认。TCP 会将它当作重发数据段的确认。这时取得的 RTT 样本反而会比较小，小于已有的 RTT。新的 RTT 反而会小于现有的 RTT，那么后面发送的数据段就更容易超时了。试验表明，在稳定状态下，将重发后收到的确认作为重发数据段的确认最终会使得计算得到的 RTT 是实际 RTT 的一半，并造成每个数据段都重发一次。

既然上述两种选择都不正确，那 TCP 协议该如何处理？答案很简单：这种情况下 TCP 就不应该进行 RTT 取样并重新计算 RTT。这个思想称为 Karn 算法，因为它是 Karn 提出来的。单纯的忽略这种情况也是不行的，因为超时的发生确实说明了网络状况的变差，TCP 应该要作出相应的反应。例如在上面提到的网络传输延时突然增大时，每个数据包都有可能会超时。如果单纯地忽略超时重发后的确认，不对超时值加以变化也同样会造成每个数据段都

要重发一次。为了反映超时重发时网络状况的变化，Karn 算法要求每次发生超时重发时 TCP 都应该增加连接的超时值。通常的做法都是将超时值增加一倍。但为了防止超时值无 限度地增加，大多 TCP 实现都规定一个超时值的上限，该值大于 Internet 上数据传输的最大延时。

Karn 算法的思想是将 RTT 的计算和超时值的计算分开。先基于 RTT 计算一个初始的超时值，然后每次发生超时重传就将增加超时值。直到下一个数据段不需要重传就收到确认时再对 RTT 取样，更新 RTT 的值，并重新基于 RTT 计算超时值。

9.4.3.4 超时值的适应范围

网络环境是动态的，会随时间变化的。尤其网络传输延时每时每刻都是不同的，它会随着网络负载的变化而变化。排队论的理论证明，RTT 的变化范围 r 会根据网络负载的不同按公式 $1 / (1 - L)$变化，这里 L 是一个不小于 0 且不大于 1 的数。上面这句话的意思是：网络的负载越重，它传输数据产生的延迟的范围就越大。例如，如果网络负载为 50%，那么网络传输延时的变化范围就是$\pm 2r$，或者 4。上面描述的计算方法是在最初的 TCP 协议中指定的，在公式 Timeout = b×RTT 中，如果将 b 设为 2 时估算的 RTT 值只能适应网络负载为 30%时的 RTT 变化。

1989 年的 TCP 规范要求 TCP 实现不但要估算 RTT 值，还要估算它的方差，并使用方差来代替公式中的常量 b 来计算超时值。这样能使得 TCP 适应的网络延时范围更广，吞吐量更高。而且方差的计算也非常简单：

DIFF = RTT 样本 － 现在的 RTT

新 RTT = 现在的 RTT + g × DIFF

新 DEV = 现在的 DEV + p× (|DIFF| − 现在的 DEV)

Timeout = 新 RTT + q× 新 DEV

上面的几个公式中，DEV 是估算出来的平均偏差，g 是一个用来控制新的样本对 RTT 值的影响程度的 0 到 1 的因子，p 是一个用来控制新样本对平均偏差的影响程度的 0 到 1 的因子，q 则是用来控制平均偏差对超时值的影响程度的因子。通常将这几个因子设为：$g = 1/8$，$p = 1/4$，$q = 3$。

9.4.4 TCP 的拥塞控制机制

所谓拥塞就是指网络中的转发设备（如路由器）因为过多的数据包到达需要转发而造成某些数据包的转发延时过大或丢失。发生拥塞时，路由器会将到达的数据包放在等待转发的队列中。需要转发的数据包越多，队列就越长，拥塞就越严重，数据包转发的延时就越长。一旦路由器的存储资源都被队列占满，那后面到达的数据包就会被丢弃。路由器在丢弃数据包时，会向数据包的源主机发送类型为源端关闭的 ICMP 数据包，但这并不会解决问题。因为拥塞造成数据包传输的延时增大甚至丢失，通常协议对此的反应就是重发数据包，而这无疑就更加重了网络拥塞的程度。拥塞程度越严重，延时就越长。这又会导致再一次的数据重发，直到网络瘫痪。因此必须在发送端的上层协议探测网络拥塞状况并对此作出反应。

在 TCP 的滑动窗口机制中说明了该机制能实现端到端的流量控制，使发送端的发送速度能适应接收端的接收能力。TCP 滑动窗口机制的另一个作用就是减轻或避免网络拥塞。

要避免或减轻网络拥塞，TCP 协议必须在网络发生拥塞时放慢数据段的发送速度。但在

控制算法上必须仔细设计，因为通常情况下网络的 RTT 也会有波动。TCP 不能为了避免拥塞而对数据传输的效率产生大的影响。TCP 用来避免拥塞的机制称为慢启动和成倍减少。这两项机制是相辅相成的。

前面提到，发送方必须根据接收方在确认段中设置的窗口字段设置发送窗口的大小。为了避免拥塞，TCP 的发送窗口的大小还受到另外一个因素的限制，这个限制称为拥塞窗口。TCP 协议实际发送窗口的大小应该是拥塞窗口和接收方接收窗口中小的一个。

我们先看成倍减少机制。在稳定状态下，没有拥塞发生，这时拥塞窗口的大小和接受方接收窗口的大小是一样的。如果发生了拥塞，TCP 可以通过减小拥塞窗口的大小来减少发送到网络的数据包的数量。TCP 对拥塞窗口的操作为：当 TCP 发现数据段丢失（即超时重发）时，就将拥塞窗口的大小减少一半（但其大小至少应为 1）。对于减小窗口后仍在发送窗口中的数据段，将其重发的超时值延长一倍。因此，如果数据段不断丢失，TCP 的发送窗口的大小会呈指数级减小，即发出的数据包会呈指数级减少，且重发的时间间隔会呈指数级增加。如果一直发生丢失，最终 TCP 发送的数据段就会减少到 1，且发送间隔会不断增加。这样显然有助于路由器从拥塞状态中恢复过来。

那么在路由器从拥塞状态恢复之后，TCP 如何恢复其正常的数据发送呢？这就需要依靠慢启动机制了。需要指出的是，在网络畅通之后，TCP 不能马上将拥塞窗口恢复到接受方窗口的同样大小，因为这样做又会使网络陷入拥塞之中。如此反复，网络会在空闲和拥塞之间波动。TCP 的慢启动机制是这样的：在 TCP 启动一个新的连接或拥塞结束后，将拥塞窗口的大小设置为一个数据段的大小。然后每收到一个确认段，TCP 就将拥塞窗口的大小增加一个数据段。慢启动机制能够避免在拥塞恢复或建立新的连接后向网络中发送的数据过快过多。虽然该机制称为慢启动机制，但如果网络通畅该机制并不会慢。因为在 TCP 收到第一个确认段后就可以发送两个数据段了，再过一个 RTT 之后就会收到两个确认段从而可以发送四个数据段了。因此在网络通畅的情况下慢启动机制的窗口增大速度是以 RTT 为时间单位随时间呈指数级增长的。为了防止慢启动后期窗口增加速度过快，TCP 协议在慢启动机制中增加了一个限制。在拥塞发生并恢复后，一旦通过慢启动机制使得拥塞窗口的大小达到了拥塞之前的一半，TCP 就启动一个拥塞避免机制以减缓窗口扩大的速度。在拥塞避免机制下，只有收到当前窗口内所有数据的确认后窗口才会增加一个数据段的大小。

9.4.5　紧急数据的传输

前面讲到 TCP 协议在发送应用程序提交的数据之前是经过缓存的。这样做的目的是为了在积累了足够的数据后组成一个长度合适的数据段进行传输，以提高数据传输的效率。这一点类似于文件操作中写文件的操作。同样，在接收方的 TCP 对于接收到的数据可能也会进行缓存，并在积累了足够的数据后再提交给上层应用程序。但有时应用程序可能有一些数据需要马上发送到连接的另一端，称这种数据为紧急数据。对于这些数据如果仍然采用这种机制显然是不合适的。

紧急数据可以分为两种。一种数据是虽然需要马上发送给连接的另一端处理，但它还是需要对方按照字节流的顺序进行处理的。因此必须先进前面的数据发送或处理后才能发送或处理该数据。例如一台主机 A 连接到另一台主机 B，主机 A 有一个终端界面接收用户的输

入，这些输入需要传输到主机 B 进行处理并返回结果后 A 才能进行下一步的工作。但用户输入的数据可能只有几个字符，显然不够一个数据段的长度。因此 TCP 需要有一种机制让用户指定马上将已经提交的数据发送出去，而不用进行缓冲。并让接收方在收到数据后马上将这些数据交给上层应用程序处理。

另一种紧急数据称为带外数据。这种数据也需要马上发送到连接的另一端进行处理。但和上面第一种不同的是，这种数据需要对方马上处理，而不管字节流的前面还有多少数据在等待处理。所以，准确地讲，带外数据是另一个数据流中的数据，和普通的数据不在一个字节流中，且带外数据的优先级更高。

9.4.5.1　强制数据传输

对于第一种紧急数据，TCP 提供了一种 push 操作。应用程序可以使用该操作要求 TCP 将缓冲区中的数据立即发送到对方。TCP 执行该操作时并不仅仅是将数据组织成数据段发送出去，它还必须使用某种手段通知接收方将接收到的数据立即提交给应用程序（即放置在某个缓冲区中，而不是放置在 TCP 内部的缓冲区中进行数据积累）。TCP 使用的方法就是将数据段的 PSH 位置 1。这样，接收方的 TCP 协议接收到该数据段后就知道应该立即将该数据提交给上层应用程序。

9.4.5.2　带外数据

带外数据的传送是通过将数据指定为紧急实现的。带外数据在 TCP 数据段中的传输以及接收端应用程序的接收方式都和普通的数据不同。当应用程序将带外数据提交给 TCP 后，TCP 将其放在一个数据段的数据区的前部，并将数据段头部的 URG 位置 1，同时将紧急指针字段设置为带外数据结束的位置。接收方的 TCP 协议收到该数据段后将带外数据从数据区提取出来，并使用操作系统特定的方式通知应用程序有带外数据到达并让应用程序立即对这些数据进行处理。这和普通数据放置在缓冲区中等待应用程序来取是不一样的。

9.5　TCP 的傻窗口症状

傻窗口症状是困扰早期的 TCP 实现的一个严重的影响 TCP 协议性能的问题。本节中先介绍傻窗口症状的特点及产生的原因，然后来看看 TCP 是如何解决这个问题的。

9.5.1　傻窗口症状

TCP 协议的滑动窗口机制实现了端到端的流量控制，使得接收方不至于因发送方的发送速度过快而被数据淹没。TCP 协议通过在确认段中提供一个窗口字段使得接受方可以将自己当前的接收窗口大小通知给发送方。接收方的做法通常是在接收到数据后就试图将数据交给应用程序处理。但应用程序可能正忙于其他事情，无暇处理这些数据。那么接收方的 TCP 协议只好减小接收窗口的大小。最后极端的情况就是接收方的应用程序一直没有处理接收到的数据，这样接收方的 TCP 缓冲区都被接收到的数据占满，接收窗口的大小减小为 0。TCP 将这个窗口大小通知给发送方 TCP。这时整个数据传输就停下来了。如果现在接收方应用程序有空处理数据并提取了部分的数据进行处理了，TCP 就会发现有部分缓冲区可用。这时

TCP 会扩大接收窗口的大小并将其通知给发送端，现在发送端又会发送该窗口大小的数据给接收方。这样整个数据的发送和接收就在这种机制下有序地进行。

但现在情况出现了。假设现在接收方的缓冲区已被数据占满，接收窗口缩小为 0，数据传输也停止了。这时，接收方的应用程序从 TCP 缓冲区中取出 1B 的数据进行处理。这时接收方的 TCP 协议发现有 1B 的缓冲区可用，就将缓冲区的大小设置为 1 并将其通知给发送方。发送方 TCP 收到该通知后就将发送窗口的大小设置为 1 并发送仅含 1B 数据的数据段给接收方。现在假设接收方以大于等于一个 RTT 的间隔每次取 1B 的数据进行处理。那么发送方每次只能发送 1B 的数据，接收方也只能接收 1B 的数据。虽然数据传输还是能够有序地进行下去，但这种状况有几个严重的问题。

首先，总是发送这种数据包浪费了网络带宽。因为在一个 TCP 段中仅包含了 1B 的数据。几乎所有的数据都是头部，极大地浪费了网络带宽。其次，发送方和接收方的 TCP 的处理代价提高了，因为他们必须为每个字节而处理一个数据段，包括对缓冲区的分配，字段的设置与检查，校验和的计算与检查等。再次，发送方和接收方底层协议（包括 IP 协议和物理层协议）的处理负担加重了，因为他们要为每个数据包或数据帧进行处理。最后，网络转发设备的处理负担也加重了。

上面这种情况因为窗口太小导致长时间发送小数据段的现象称为傻窗口症状。产生傻窗口症状的原因并不只上面提到的一种。发送方如果处理不适当也会引起这种现象。例如，如果发送端的 TCP 协议不对上层应用程序提交的数据进行缓冲，在应用程序每次只提交 1B 的数据时也会产生傻窗口症状，因为它会立即将这个字节的数据组织成一个数据段发送出去。

9.5.2　傻窗口症状避免机制

TCP 协议在接收方和发送方都有一套机制来避免产生傻窗口机制。发送方的机制主要用来防止在数据段中只包含少量数据，而接送方的机制用来防止发送小的窗口通知给发送方。实际上，在一个连接中只需要有其中一方的机制就可以避免傻窗口症状了，但在实现时，TCP 协议要求发送方和接收方都要实现各自的机制。这样要求是为了防止实际通信中某一方不按照该机制行为。所以一个 TCP 协议的具体实现中应该都包含这两套机制，因为 TCP 的数据传输是全双工的，任何一方既会发送数据也会接收数据。

9.5.2.1　接收方的傻窗口症状避免机制

为了防止向发送方发送小的窗口通知，接收方在向发送方发送过 0 大小的窗口通知后，如果应用程序处理数据的速度较慢，使得产生的可用缓冲区太小，TCP 不会马上将窗口扩大至该缓冲区的大小，也不发送窗口改变通知。它会继续等待应用程序处理数据，直到可用的缓冲区足够大之后再扩大窗口并向发送方发送窗口通知。那么，可用缓冲区多大才算足够大呢？这要根据接收方的缓冲区的最大大小和最大数据段长度决定。TCP 协议将足够大定义为最大可用缓冲区大小的一半和最大数据段长度中小的一个。

接收方有两种方法可以实现上述的这种机制。第一种方法是接收方对收到的数据及时进行确认，但如果发现可用窗口的大小不能满足足够大的要求就发送窗口扩大通知。第二种方法是在窗口大小没有达到可以发送窗口扩大通知的大小时延迟对收到数据的确认。TCP 协议的标准推荐使用第二种方法。

延迟对接收数据的确认可以减少网络上不必要的流量。首先，它可以减少确认段的数量，因为在这段延迟的时间内有新的数据到达，TCP 就可以使用一个确认段对所有的数据进行确认了。其次，在这段时间内可用缓冲区的大小可能会变得足够大，因为应用程序在不断地处理数据，可用缓冲区随时有可能会增加。如果是这样，那么窗口扩大通知就可以和确认在一个数据段中发送了。

但延迟数据确认的发送也有它的缺点。如果接收方延迟的时间太长，发送方在超时后就会重发数据。重发的数据不但浪费了网络带宽，还无谓地增加了连接两端的处理负担。另外，由于 TCP 要使用确认到达的时间来估算连接的 RTT，因此延迟确认的发送会增加 RTT 的大小并最终导致对丢失数据的重发等待时间太长，降低数据传输的效率。

为了避免延迟对数据的确认的发送，TCP 标准规定对数据确认的发送的延迟不能超过 500ms。另外，为了让发送方 TCP 能收集足够的 RTT 样本，TCP 规定对其他情况下的数据段必须逐一进行确认。

9.5.2.2 发送方的傻窗口症状避免机制

因为发送方的应用程序有可能会提交少量的数据给 TCP 协议进行传输，如果 TCP 立即将这些数据组织成数据段发送出去就会出现傻窗口症状。因此，TCP 必须将应用程序提交的数据进行缓存，等积累到足够多的数据时再组织数据段发送出去。现在的问题是 TCP 协议应该等待多久？因为 TCP 协议不知道应用程序是否还有数据需要传输，如果等待时间太长，应用程序可能会觉得延时太长；等的时间太短，就会产生过小的数据段。实际上，TCP 协议根本不会计时，它使用的策略是：如果应用程序向 TCP 协议提交了少量数据进行传输，而在此之前 TCP 已经向接收方发送过数据但还没有收到确认，TCP 就将应用程序提交的数据暂存在本地缓冲区中，并不发送，直到积累的待发送的数据足够组成一个最大数据段长的数据段。或者在收到以前数据的确认时积累的数据还不够多，TCP 就不再等待，将缓冲区中的数据发送出去。即使对请求了 push 操作的数据也应用该策略。

该策略称为 Nagle 算法，是根据其发明人命名的。该算法具有以下几个优点：首先，它几乎不需要发送方为该算法付出额外的处理。它不需要为每个连接启动一个定时器。其次，它对各种应用都适用，不会影响它们的吞吐量和反应速度。

9.6 TCP 协议与 UDP 协议的比较

很显然，TCP 协议比 UDP 协议复杂得多。在本节中将两者做一个简单的比较，主要是从协议的应用范围说明它们的适应范围。

9.6.1 TCP 协议与 UDP 协议特点的比较

首先，TCP 是一种面向连接的协议，而 UDP 是无连接的协议。这其中的区别在于：第一，TCP 协议是以连接作为协议数据的最终目标的。UDP 协议则是以目标端口作为协议数据的最终目标。因此，TCP 的协议端口是可以复用的，UDP 协议的端口在同一时间则只能为一个应用程序所用。第二，一个连接是由两个端点构成的。要使用 TCP 进行通信必须先在通信

双方之间建立连接，连接的两端必须就连接的一些问题进行协商（如最大数据段长度、窗口大小、初始序列号等），并为该连接分配一定的资源（缓冲区）。UDP 协议则不需要这个过程，可以直接发送和接收数据。

其次，TCP 提供的是可靠的传输服务，而 UDP 协议提供的是不可靠的服务。使用不可靠的服务进行数据传输时，数据可能会丢失、失序、重复等。而可靠的服务能保证发送方发送的数据能原样到达接收方。

最后，TCP 提供的是面向字节流的服务。应用程序只需将要传输的数据以字节流的形式提交给 TCP 协议，在连接的另一端，数据以同样的字节流顺序出现在接收程序中。而 UDP 协议的传输单位是数据块，一个数据块只能封装在一个 UDP 数据包中。

9.6.2　TCP 协议与 UDP 协议应用的比较

根据以上分析的 TCP 协议和 UDP 协议的特点，我们来看看它们各自在哪些环境中应用。

因为 TCP 协议提供了可靠的面向字节流的服务，而且有一套高效的机制保证数据的高效传输，所以对于有大量数据需要进行可靠传输的应用是很适合的，因为应用程序无需关心如何保证数据传输的可靠性，如何进行超时重发等。这种应用的典型例子就是文件传输协议（FTP）。

由于 TCP 协议要先建立连接之后才能进行通信，而连接的建立过程需要一定的时间。所以如果应用程序只有少量数据（例如可以在一个数据包中进行封装）需要传输则不适合使用 TCP 协议，因为连接建立的开销大于其方便性的优点。但对于虽然数据量少但需要时间较长且可靠性要求高的应用 TCP 也是比较适合的。Telnet 就是这种应用的一个例子。

实时应用不管数据量大小，不管对可靠性要求高低都不适合使用 TCP 协议，因为 TCP 协议对数据的传输是有先后顺序的，只有前面的传输成功才会开始后面的数据传送。这显然是不符合实时应用的要求的。

另外，由于 TCP 协议是面向连接的，一个连接必须且只能有两个端点。所以对于多个实体间的多播式应用无法使用 TCP 进行通信，因为对于 n 个实体间的通信需要 $n \times (n - 1) / 2$ 个连接。n 很大时连接数太多。

对于不适合使用 TCP 协议的应用就只能使用 UDP 协议了。但使用 UDP 协议进行通信时应用程序必须自己处理下列问题：

（1）应用程序必须自己提供机制来保证可靠性。应用程序必须有自己的超时重发机制、数据失序的处理、流量控制等。当然对于一些可靠性要求不高的应用可以不用这些机制，但通常都需要区分数据的先后关系。

（2）应用程序必须自己处理大块数据的分割，以让其能封装在一个 UDP 数据包中。在接收方还必须再将分割的数据进行重组。

9.6.3　常见的标准 TCP 协议端口

表 9.1 是常见服务的标准 TCP 端口及其服务简要说明。

表9.1　标准TCP端口及其说明

端　口　号	描　　述
0	保留
1	TCP 多路复用器
5	远程任务入口
7	回显
9	丢弃
11	活动用户
13	日期时间
15	netstat
17	qotd
19	字符产生服务
20	FTP 数据传输
21	FTP
23	TELNET
25	SMTP
37	时间
42	主机名服务器
43	whois
53	域名服务器
79	Finger
93	设备控制协议
101	NIC 主机名服务器
103	网络时间协议
103	X.400 邮件服务
104	X.400 邮件发送
113	认证服务
119	USENET 新闻传输协议
139	NETBIOS 会话服务

第 10 章　远程登录

在 ARPANet 建设的初期，计算机尚属昂贵的资源。当时 PC 机尚未普及，计算机大多是运行多用户操作系统的中小型机。这些中小型机通常由一台主机和多个终端组成，主机的计算资源在多个终端用户间共享，系统为每个用户分配一账号，账号规定了用户对系统的访问权限。用户用自己的账号在系统的某一终端登录后便可以访问系统的部分或全部资源。ARPANet 出现以后，远端用户利用智能终端通过 ARPANet 远程登录计算机系统以共享异地计算资源的需求便相应产生，远程登录协议即是为满足这一需求而设计的。

远程登录的根本目的在于远端用户可以像本地用户一样访问远地系统的资源。因此远程登录协议设计的目标是提供一个相对通用的、双向的、面向 8 位字节的通信方法，使得远端的终端设备能以标准的方法与计算机系统进行交互作用。目前 TCP/IP 协议族中有两个远程登录协议：Telnet 和 rlogin。Telnet 协议由 RFC854 定义了其规范，是 Internet 上主机被要求采用并实现的标准。rlogin 是 Sun 微系统公司专门针对 BSD UNIX 系统而设计的远程登录　协议。

本章主要讨论 Telnet 协议的服务方式、工作原理以及 Telnet 协议的主要命令，并简要介绍 rlogin 协议。

10.1　远程登录的服务模式

远程登录协议运行于 TCP 协议上，客户终端与远地计算机采用 TCP 连接进行通信。远程登录服务采用客户-服务器模式。如图 10.1 所示，在终端处运行有 Telnet 客户进程，远地计算机系统处有一服务器进程，客户进程和服务器进程通过一条 TCP 连接进行交互。

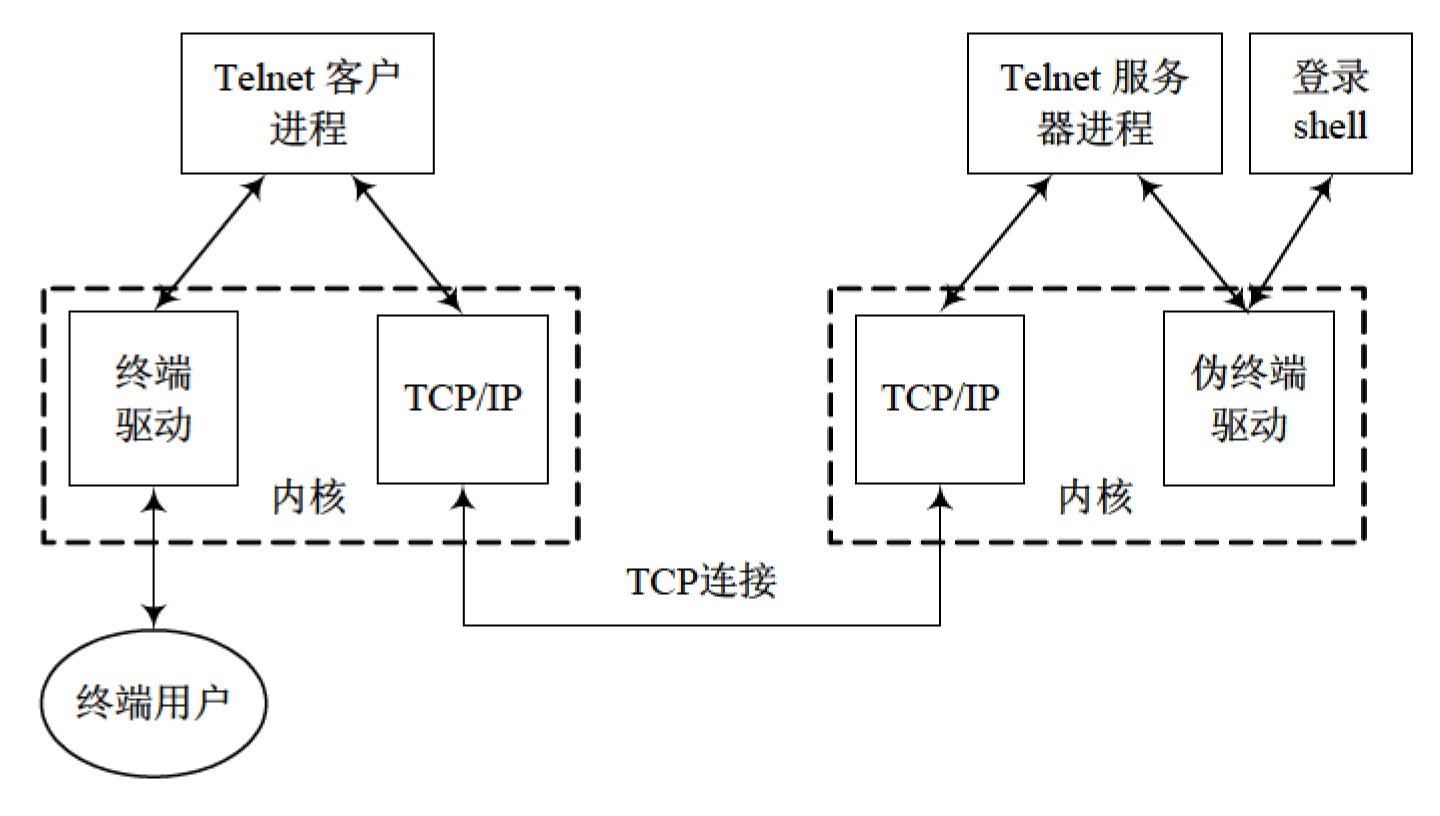


图 10.1　远程登录的服务模式

这里要注意的是，远程登录服务的目标是为远端用户提供与本地用户相同的服务。由于一般系统提供给本地用户的命令至少有几十条，因此系统同时要提供给远端用户这些命令的服务。这时若单纯地使用客户-服务器模式就必须为每一登录用户的每一条可能使用的命令建立一个进程，这导致系统为接纳一个远程登录必须创建至少几十个服务器进程，使得系统的扩展性差。远程登录服务解决这一问题的方案是为每个远程登录的用户只创建一个静态的 shell 进程，远端用户的服务请求均由 shell 进程进行解释处理，然后动态地 fork 出相应的进程完成具体的服务。这样大大减少了系统中静态进程的数目，避免了系统进程数随登录用户数迅速膨胀，有效解决了上述的扩展性问题。因此，Telnet 远程登录在连接的双方各只需一个应用程序实现，客户终端处运行的程序是客户进程，远地计算机运行的是服务器进程。

远地提供远程登录服务的主机中始终运行有 Telnet 服务器进程，随时接收终端用户的登录请求（这类进程也称为守护进程）。一次远程登录的过程是由终端用户调用 Telnet 命令开始的。此时，终端处与 Telnet 命令相对应的应用程序称为客户。然后客户与远地计算机上的远程登录服务器建立 TCP 连接，远程登录服务器在接收到连接请求后为该用户创建一个 shell 进程。在此 TCP 连接基础上，客户将从用户终端接收的键盘输入传给服务器端的 shell 进程。shell 进程在接收到终端用户的命令后，对其进行解释并 fork 出一子进程运行相应的程序。命令执行的结果通过 TCP 连接返回到客户处。客户再将服务器返回的字符通过终端处的操作系统将它显示在用户终端上。

从图 10.1 可以看出的另一点是在 Telnet 服务中服务器进程和客户进程均未放入操作系统内核，这主要是基于两个考虑：① 远程登录主要是服务于键盘类的终端，微秒级的进程切换所造成的响应延迟与人的击键速度相比是微不足道的，因此将 Telnet 服务放在应用程序级并不影响它的正常工作。② 将 Telnet 类的程序作为应用程序可以使系统的内核更紧凑简洁，使得系统具有良好的开放性和可维护性。

10.2　Telnet 原理

从 10.1 节的介绍知道 Telnet 远程登录服务过程分为三个步骤：

第 1 步，远端用户在终端上对系统进行远程登录。该远程登录的内部视图实际上是一个 TCP 连接。

第 2 步，将远地终端上的键盘输入逐键传到主机系统。

第 3 步，将主机系统的输出送回本地系统。

这一过程看似简单，但 Internet 上主机和终端千差万别，各自有着不同的生产厂家，使用了不同的技术，要使 Telnet 能在 Internet 上正常工作尚需使输入/输出对远端系统内核透明。Telnet 通过网络虚终端和选项协商机制来实现这种透明性。

10.2.1　网络虚终端（NVT）

对于 Telnet 远程登录来说，系统间的异构性表现在什么地方呢？表现在不同的系统对键盘输入的解释各不相同。比如行结束标志，当按下回车键时，所有的系统都会换行，这是相同的；不同的是，有些系统以 ASCII 字符 CR（Carriage Return）作为行结束标志，而有些系

统则又以 CR-LF 两个字符作为行结束标志。以不同字符作为行结束标志的系统显然不能直接进行远程登录。又如，用于产生进程终止的键码也可能随系统而不同。有的系统以 Ctrl+C 键作为终止码，而有的系统以 Del 键作为终止符。再比如，异构系统之间流控字符也可能有差异等。为了统一异构系统对键盘输入的解释，Telnet 专门提供一种标准的键盘定义方式，叫做网络虚终端（NVT，Network Virtual Terminal）。

1．NVT 原理

Internet 中解决异构性的基本思想是各种与设备相关的数据格式映射到某个约定好的且与设备独立的数据格式，以封装屏蔽设备的异构性。正如 IP 协议用来屏蔽不同物理子网的异构性那样，Telnet 采用 NVT 来屏蔽不同终端的不同物理特性。NVT 的原理如图 10.2 所示。

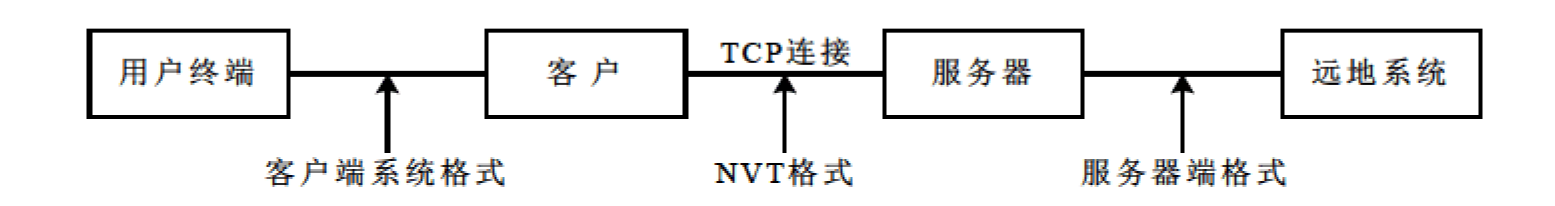


图 10.2　NVT 原理

在客户和服务器两端，输入/输出采用各自的本地格式。在远程登录连接上，客户软件将终端用户输入转化为标准的 NVT 数据和命令序列，经 TCP 连接传到远地机上的服务器，服务器再将 NVT 序列转化为远地系统的内部格式。由于客户和服务器既了解各自系统的内部格式，又了解 NVT 定义，所以上述转换很容易实现。在 NVT 的作用下，不同的本地格式得以统一起来，在 NVT 一层实现了一致性。各本地格式只跟标准的 NVT 打交道，而不跟其他具体格式直接互操作。这样不必针对每个具体的远地终端或本地机而编写客户或服务器程序，使客户或服务器程序需针对宿主系统开发，具有很大的通用性，简化了程序设计，从而为实现异构系统的互操作打下了基础。

2．NVT ASCII 码

术语 NVT ASCII 代表 7 比特的 ASCII 字符集，网间网协议族都使用 NVT ASCII。每个 7 比特的字符都以 8 比特格式发送，最高位比特为 0。

在标准 ASCII 码的 128 个字符中，有 95 个为可见字符（包括字母、数字、标点符号和一些特殊图形字符），NVT 保持这些字符的原来意义；另外，有 33 个字符为控制码，NVT 对其中的 8 个进行了重定义，如表 10.1 所示。

表10.1　NVT对ASCII控制码的重定义

ASCII 控制码	数　值	NVT 意义
NUL	0	无操作（对输出无影响）
BEL	7	发声光信号（光标不动）
BS	8	左移光标一格
HT	9	水平右移到下一 Tab 位置
LF	10	垂直下移一行
VT	11	垂直下移到下一 Tab 位置
FF	12	移到下一页头
CR	13	移到本行的左端

行结束符以两个字符 CR（回车）和紧接着的 LF（换行）这样的序列表示，以\r\n 来表示。单独的一个 CR 也是以两个字符序列来表示，它们是 CR 和紧接着的 NUL（字节 0），以\r\0 表示。

许多系统给终端用户提供一些简单的控制命令，比如 BSD Unix 中的 Ctrl+C 用于终止当前进程的执行。控制命令不是正常的程序输入，不为应用程序所接受，而应由操作系统解释。NVT 如何表示控制命令？NVT 设想在用户键盘上有一组虚拟键，用以产生这类控制命令。NVT 一共定义了 7 个虚拟键，如表 10.2 所示。

表10.2　NVT虚拟键

信　　号	命　　令
IP	终止当前进程
AD	抛弃当前输出
AYT	服务器测试
EC	删除前一字符
EL	删除当前行
SYNCH	带外信号
BRK	停止

这类 NVT 命令一般由操作系统解释，在传输时应将它们和 Telnet 数据流区分开来。即采用所谓的带外信号传输。为什么要采用带外信号传输呢？我们以终止进程命令 IP（Interrupt Process）为例来说明。IP 命令一般用在远地进程发生错误动作而本地用户想终止该进程的时候。假如远地进程进入一个死循环而又不能读写任何数据，在其输入缓冲区满后，客户再也写不进任何信息到虚终端，当然也就写不进 IP 命令。一旦服务器无法接收 IP 命令，这远地进程将无休止地运行下去，直至远地系统关机。因此，为避免这类故障，NVT 控制命令必须采用带外信号传输。

当 Telnet 将控制命令放入数据流时，它首先发送一个 SYNCH 命令，然后再发送控制命令，最后还要加一个 DMARK 命令（见 10.2.2 节）。这样就将控制命令与一般数据分开。另外，要将带外信号传给服务器还要借助于 TCP 的 URGENT 机制（紧急数据），因为假如 TCP 连接因为远地系统缓冲已满而不接收数据，则控制命令还是传不到服务器。

TCP 将传输带外信号的 TCP 报文的“紧急数据”位置 1，该段便会不受流控的限制立即传到服务器。服务器收到紧急信号后，读入并抛弃所有数据，直到发现 DMARK 标志。然后服务器再转入正常工作，将控制命令交给操作系统执行。

10.2.2　Telnet 命令

Telnet 通信的两个方向都采用带内信令方式，即 Telnet 命令放入数据流中传输。这就需要能将 Telnet 命令同普通数据区分开来。Telnet 采用了转义序列的办法。每个转义序列由两个字节构成，前一个字节是 0xff（十进制的 255）叫做 IAC（interpret as command，意思是“作为命令来解释”），其作用是指出该字节后面的一个字节是命令字节。如果要发送数据 255，就必须发送两个连续的字节 255。将 Telnet 命令与标准 ASCII 码分开的优点是增加了 Telnet

的灵活性，使 Telnet 可以传输所有可能的字符序列和命令序列。表 10.3 给出了所有的 Telnet 命令。

表10.3 Telnet命令集

命 令	十进制编码	意 义
IAC	255	转义序列开始符
DON'T	254	拒绝执行指定选项的请求
DO	253	批准开放指定选项
WON'T	252	拒绝执行指定选项
WILL	251	同意执行指定选项
SB	250	选项协商开始
GA	249	继续进行
EL	248	删除行
EC	247	删除前一字符
AYT	246	对方是否在运行
AO	245	异常终止输出
IP	244	中断进程
BRK	243	中断
DMARK	242	数据标记
NOP	241	无操作
SE	240	选项协商结束
EOR	239	记录结束符
ABORT	238	异常终止进程
SUSP	237	挂起当前进程
EOF	236	文件结束符

10.2.3 选项协商

在 Telnet 中，系统的异构性除了终端对键盘输入解释的区别外，不同的终端系统在功能等方面也存在着种种差别。如有些终端只能工作于半双工方式，而一些终端却可以在全双工下工作，NVT 对屏蔽系统间的这类差异是无能为力的。Telnet 提供了选项协商机制来解决这一问题。

Telnet 连接的双方在建立起 NVT 通信之前，首先交互选项协商数据。选项协商是对称的，也就是说，任何一方都可以主动发送选项协商请求给对方。选项协商需要 3 个字节：一个 IAC 字节，接着一个字节是选项协商命令，最后一个字节是本次协商要激活或禁止的具体选项 ID。

1．Telnet 选项

现在，有 40 多个选项是可以协商的。AssignedNumber RFC 文档中指明选项字节的值，并且一些相关的 RFC 文档描述了这些选项。表 10.4 显示了一些常用的选项代码。

Telnet 的选项协商机制和 Telnet 协议的大部分内容一样，是对称的。连接的双方都可以发起选项协商请求。但我们知道，远程登录不是对称的应用。客户进程完成某些任务，而服

务器进程则完成其他一些任务。下面将看到，某些 Telnet 选项仅仅适合于客户进程（例如要求激活行模式方式），某些选项则仅仅适合于服务器进程。

表10.4　常用的Telnet选项

选项标识	名　称	RFC
1	回显	857
3	抑制继续进行	858
5	状态	859
6	请求时间标志插入返回流	860
24	交换关于终端的型号信息	1091
31	窗口大小	1073
32	终端速率	1079
33	远程流量控制	1372
34	行方式	1184
36	环境变量	1408

2．选项协商命令

对于任何给定的选项，连接的任何一方都可以发送下面 4 种请求的任意一个请求。

（1）WILL：　发送方本身将激活（enable）选项。

（2）DO ：发送方想叫接收端激活选项。

（3）WONT ：发送方本身想禁止选项。

（4）DON'T：发送方想让接收端去禁止选项。

由于 Telnet 规则规定，对于激活选项请求，如（1）和（2），有权同意或者不同意。而对于使选项失效请求，如（3）和（4），必须同意。这样，4 种请求就会组合出 6 种情况，如表 10.5 所示。

表10.5　选项协商命令的6种组合

	发　送　方	接　收　方	描　　述
1	WILL		发送方想激活选项
		DO	接收方说同意
2	WILL		发送方想激活选项
		DON, T	接收方说不同意
3	DO		发送方想让接收方激活选项
		WILL	接收方说同意
4	DO		发送方想让接收方激活选项
		WONT	接收方说不同意
5	WONT		发送方想禁止选项
		DON, T	接收方必须说同意
6	DON, T		发送方想让接收方禁止选项
		WONT	接收方必须说同意

3．子选项协商

有些选项不是仅仅用“激活”或“禁止”就能够表达的。指定终端类型就是一个例子，客户进程必须发送一个 ASCII 字符串来表示终端类型。为了处理这种选项，我们必须定义子选项协商机制。在 RFC1091[VanBokkelen 1989]中定义了如何表示终端类型这样的子选项协商机制。

首先连接的某一方（通常是客户进程）发送 3B 的字符序列<IAC, WILL, 24>来请求激活该选项。如表 10.4 所示，这里的 24 表示终端类型选项。如果接收端（通常是服务器进程）同意，那么响应数据是：

<IAC, DO, 24>

然后服务器进程再发送如下的字符串：

<IAC, SB, 24, 1, IAC, SE>

该字符串询问客户进程的终端类型。其中，SB 是子选项协商的起始命令标志。下一个字节的“24”代表这是终端类型选项的子选项（通常 SB 后面的选项值就是子选项所要提交的内容）。下一个字节的“1”表示“发送你的终端类型”。子选项协商的结束命令标志也是 IAC，就像 SB 是起始命令标志一样。

如果终端类型是 ibm pc，客户进程的响应命令将是：

＜IAC, SB, 24, 0‘I’, ‘B’, ‘M’, ‘P’, ‘C’, IAC, SE＞

第 4 个字节“0”代表“我的终端类型是”。在 Telnet 子选项协商过程中，终端类型用大写表示，当服务器收到该字符串后会自动转换为小写字符。

4．选项协商示例

对于大多数 Telnet 的服务器进程和客户进程，共有半双工、一次一字符、一次一行和行方式等 4 种操作方式。其中一次一字符方式为大多数 Telnet 程序的默认工作方式。一次一字符方式是指用户在终端输入的每个字符都将由终端发送到服务器进程，服务器进程的响应也将以字符方式回显到终端上。下面我们以一次一字符方式为例说明 Telnet 的选项协商过程。

假设客户端运行的是较新版本的 Telnet 程序，而服务器端运行的 Telnet 版本较旧。客户端在建立 Telnet 连接时试图激活较多的选项，服务器由于不支持其中某些选项而加以拒绝。整个协商过程如图 10.3 所示。

图 10.3 中协商过程的含义如下：

（1）客户发起 SUPPRESS GO AHEAD 选项协商。由于 GO AHEAD 命令通常是由服务器发送给客户的，而且客户希望服务器激活该选项，因此该选项的请求方式是 DO。服务器进程的响应是 WILL SUPPRESS GO AHEAD，即同意该选项。

（2）客户进程要按照在 RFC 1091[VanBokkelen 1989]中的定义发送终端类型。因为客户进程要激活本地的选项，所以该选项的请求方式是 WILL。服务器进程同意激活终端类型选项。但现在客户进程还不能立即发送它的终端类型。它必须要等到服务器进程用子选项的形式询问终端类型的时候才能够发送。

（3）NEWS 的意思是“协商窗口大小”，它在 RFC 1073 [Wa i t z m a n]中有定义。WILL NEWS 是指客户端要求协商窗口大小，服务器回应 DON’T NEWS 加以拒绝。

（4）TSPEED 选项允许发送方（通常是客户进程）发送它的终端速率，这在 RFC 1079[Hedrick 1988b]中有定义。如果服务器进程同意，客户进程将发送其发送速率和接收速

率的子选项。服务器发送 DON'T TSPEED，拒绝该请求。

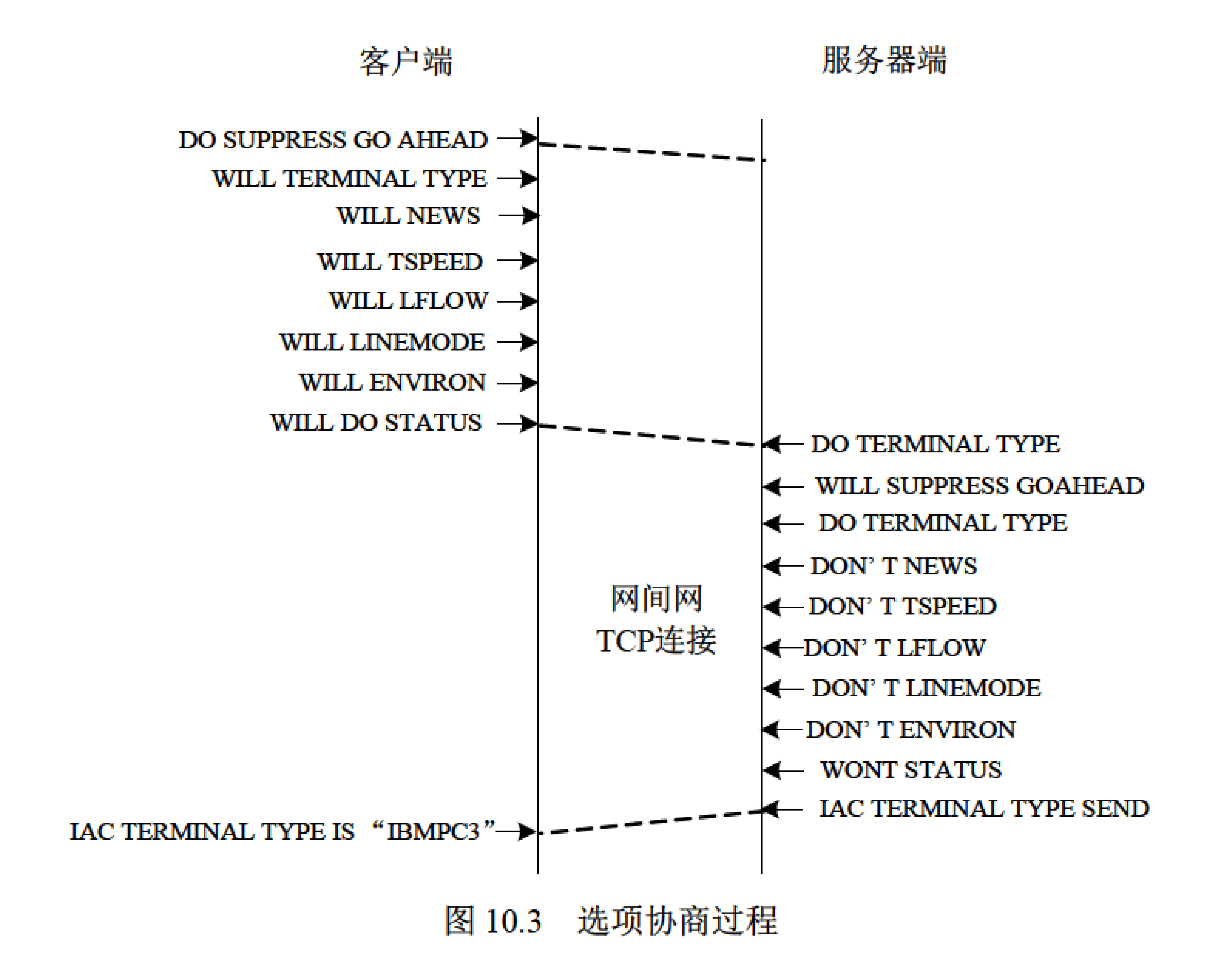


图 10.3　选项协商过程

（5）LFLOW 代表"本地流量控制"，这在 RFC1371 [Hedrick 和 Borman 1992]中定义。客户进程给服务器进程发送该选项，表示客户进程希望用命令方式激活或禁止流量控制。本例中服务器不支持该请求。

（6）LINEMODE 代表实行方式。所有终端字符的处理由 Telnet 客户进程完成（例如回格、删除行等），然后整行发送给服务器进程。该选项同样被服务器进程拒绝。

（7）ENVIRON 选项允许客户进程把环境变量发送给服务器进程，这在 RFC 1408 [Borman 1993a]中有定义。这样就可以把客户进程的用户环境变量自动传播到服务器进程。服务器进程拒绝该选项。

（8）STATUS 选项（RFC 859 [Postel 和 Reynolds 1983e]中定义）允许连接的一方询问对方对 Telnet 选项目前状态的理解。客户进程要求对方激活选项（DO）。如果服务器进程同意客户进程就可以要求服务器进程以子选项的形式发送它的状态值。在这个例子中，服务器拒绝了该请求。

（9）服务器发送 IAC TERMINAL TYPE SEND 要求客户进程发送终端类型子选项；客户进程把终端类型"IBMPC3"以 6B 的字符串形式发送给服务器进程。

10.3　rlogin

rlogin 的第一次发布是在 4BSD 中，当时它仅能实现 Unix 主机之间的远程登录。这就使得 rlogin 比 Telnet 简单得多。由于客户进程和服务器进程的操作系统预先都知道对方的操作系统类型，不必太考虑异构性处理，所以就不需要选项协商机制。rlogin 支持以下功能：

（1）信任主机

管理员可以选出一组主机，叫做信任主机，赋予它们共享资源的特权。通过 rlogin，在信任主机之间可以共享登录名（即用户账号）和文件访问权限，而且任何登录（本地或远程）都是平等的。

（2）自动授权机制

用户可以把自己账号的访问权赋予远程登录。比如，本地机 X 用户可以允许远地机 H 上的 Y 用户访问自己的账号。这样，假如 Y 在 H 上登录后，再通过 rlogin 登录进入 X 账号，便不需再输入口令。

（3）上述自动授权机制还可用于通用程序执行

比如，假设主机 H1 上的 X 用户将自身账号的访问权赋予 H2 主机上的 Y 用户，则 Y 可以直接在 H2 上执行 X 也可以执行的 H1 上的命令：

H2.Y>rsh　H1　command [parameter]

Rsh 是 rlogin 的变形，它表面上省略了 rlogin 的过程，由内部隐藏执行。Y 执行 rsh 实际上分为两个步骤。

第 1 步，隐藏执行：

rlogin　H1　X

第 2 步，执行：

command [parameter]

其中，由于第 1 步中 X 已对 Y 授权，所以不会出现“password”提示符，也就不必再输入 Y 的口令。

另外，rsh 的执行效果相当于直接在远地机上执行一条命令，标准输入来自本地机，标准输出去往本地机。如同本地登录一样，rlogin 和 rsh 也可进行输入/输出重定向及流控等操作。

第 11 章　电子邮件

电子邮件是 Internet 上的最早应用之一，最早出现在 Internet 前身 ARPANET 上。由于电子邮件与传统邮件相比具有传递快速，风雨无阻的优点，同时又不像电话那样需要通信双方同时在场，因此电子邮件应用出现后发展迅速，目前已成为 Internet 上最流行的应用之一。有资料表明，TCP 连接的一半以上是用于简单邮件传输协议的。现有的各种网络体系结构也无一例外地把电子邮件作为一个重要的应用，纳入自己的协议族。

本章主要介绍 TCP/IP 电子邮件系统结构原理以及 TCP/IP 的电子邮件协议。

11.1　电子邮件系统结构

传统的邮件服务对我们来说是再熟悉不过的了，其工作流程在第 3 章已有说明。电子邮件作为传统邮件业务在网络时代的衍生物也有着类似的工作过程。图 11.1 给出了 TCP/IP 邮件系统的示意图。

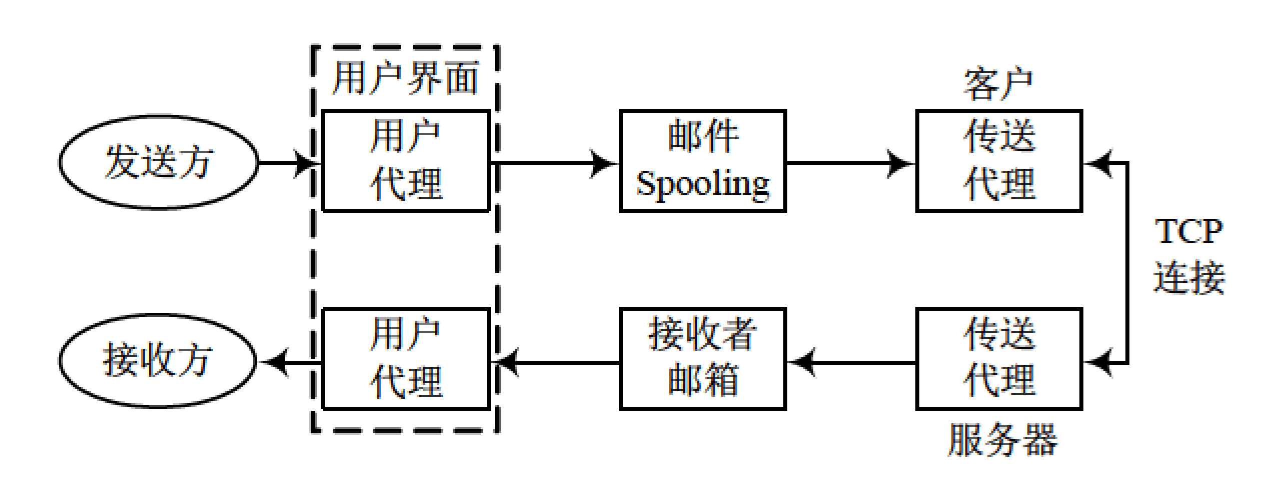


图 11.1　TCP/IP 邮件系统示意图

无论是传统邮件业务还是电子邮件系统，需要解决的首要问题是邮件的传输。传统的邮件是由从发信邮局到收信邮局沿途的邮局接力传递的。TCP/IP 电子邮件系统与之有所不同。这是由于 TCP/IP 自始至终坚持端到端的思想，它的电子邮件系统也不例外地采用了端到端的传输方式。在 TCP/IP 电子邮件系统中负责邮件传输的部分称作邮件传送代理。邮件传送代理采用客户机/服务器模式工作，发送方的传送代理作为客户机，接收方的代理作为服务器。在邮件传输时由发送方发起连接请求，服务器响应后建立起 TCP 连接，邮件信息便在此 TCP 连接上进行传输。在发送方的传送代理和接收方代理之间一般不进行邮件的存储和转发，这是 TCP/IP 电子邮件系统与 ISO/OSI 电子邮件系统的最大区别，后者是采用存储转发方式工作的，更类似于传统的邮件业务。

除了邮件传输外，电子邮件系统还必须解决如下问题：

（1）邮件编辑

该功能是帮助用户编写邮件并为邮件填写地址和其他控制信息。

（2）转换及格式化

转换是指将信息转换成适合于在接收者终端上显示或打印的格式。格式化是指解决邮件在接收者终端上的格式化显示问题。对邮件显示格式的最直接处理方式是：电子邮件系统传来未格式化邮件，由用户调用格式化程序进行处理，再调用显示程序（如编辑器）对格式化文件进行阅读。这种处理方式对无经验的用户是很困难的。最好是电子邮件系统能直接显示格式化邮件，这样用户操作就大大简化了。

（3）邮件处置

帮助接收者处理所收到的邮件，包括立即扔掉、读完扔掉、读后保存、转发邮件和阅读旧邮件等操作。

上述这些功能均由用户代理来实现。用户代理是用户使用邮件系统的界面，它不仅可以帮助用户完成编辑邮件、显示和处置邮件等工作，同时还隐藏了邮件传输的复杂性。即用户在使用邮件系统时，只需与用户代理打交道而无需了解传送代理工作过程。这大大方便了用户对邮件系统的使用。

在图 11.1 所示的系统中，还有两个部分：接收者邮箱和邮件发送的 Spooling 区。其中接收者邮箱的含义和功能是不言自明的，它是邮件传送的目的地。每个邮件系统的使用者都至少拥有一个邮箱，邮箱是私有的，电子邮件系统的软件可以往邮箱中添加一封邮件，但只有邮箱拥有者才能检查或删除该邮件。而邮件发送的 Spooling 区是发送邮件的缓存队列。Internet 上的主机可能会由于故障或网络拥塞等原因导致暂时的无法访问，设立发送邮件缓存区主要就是解决接收方主机暂时不能访问时，如何发送电子邮件的问题。用户通过用户代理写好邮件，提交发送后邮件即进入缓存区。发送方传送代理将周期性地检查发送缓冲区，每当它发现未发邮件，或用户传来一个新邮件，传送代理立即着手发送。当发现某邮件很长时间都发不出去，传送代理将它返回发送者。

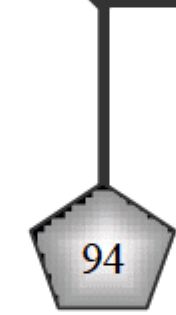
11.2　TCP/IP 电子邮件地址

电子邮件地址即邮箱地址，Internet 上每个邮箱必须有惟一的地址。TCP/IP 电子邮件系统的编址采用了两层的方案，即每个地址由两部分组成，两部分之间用“@”隔开。其格式如下：

local-part@domain-name

第一部分为本地名，即邮箱名，它可以是用户起的任意名字，只要不与同一邮件服务器上的其他邮箱重名。“@”后的为第二部分，它是邮件服务器所在主机的域名。由于域名是全网惟一的，因此这种编址方案保证了每个邮箱的地址也是惟一的。

另一方面，采用两层的编址还有两个优点。第一，这种划分允许每个计算机系统规定自己的邮箱的标识，不同的计算机可以使用不同的邮箱标识机制，也可以使用相同的邮箱标识机制。第二，这种划分允许任意计算机系统上的用户交换电子邮件信息。发送方计算机上的电子邮件软件在发送信息时使用地址中的第二个部分来确定要连接的计算机。接收方的计算机上的电子邮件软件使用地址中的第一个部分来选择邮箱将信息放进去。这样，地址的第一个部分是本地翻译的，该字符串在一个计算机系统之外没有任何意义。



11.3　电子邮件格式

11.3.1　电子邮件信息格式

电子邮件信息的格式很简单，由 RFC822 定义了 TCP/IP 电子邮件的信息格式。信息由 ASCII 文本组成，包括两个部分，中间用一个空行分隔。第一部分是一个头部（header），包括有关发送方、接收方、发送日期和内容格式等控制信息。第二部分是正文（body），包括信息的文本。

虽然信息的正文可以包含任意文本，但电子邮件软件在收发信息时仍使头部保持标准形式。每个头部行首先是一个关键字，一个冒号，然后是附加的信息。关键字告诉电子邮件软件如何翻译该行中剩下的内容。有些关键字在电子邮件头部是必需的，另一些是可选的。例如，每个头部必须包含以 To 开头的行，说明一个接收方的列表。这行中 To 和随后的冒号之后的内容包含了一个或多个电子邮件地址，每个地址对应一个接收方。电子邮件软件在电子邮件的头部放置一个以 From 开头的行，其后跟随的是发送方的电子邮件地址。图 11.2 是一个电子邮件信息的实例，以说明头部行的形式。

```
From: wang@sina.com
To: zhang@163.net
Date: Fri，18 Dec 2003 12:30:30
Subject: 你好吗？

老张，
好久未联系，你近来可好？有空来我这儿好好聊聊。
老王
```

图 11.2　一个邮件实例

图中显示了两个附加的行，包括信息发出的日期及信息的主题。这两个都是可选的，发送方的电子邮件软件选择是否包含它们。除了上例中的关键字外 RFC822 还定义了其他一些关键字，如关键字 Cc 用来表明邮件副本的抄送地址，Reply-To 用来表明邮件的回复地址，X-Mailer 用来指明发送邮件所使用的软件等。

11.3.2　多用途互联网邮件扩充

最初的因特网电子邮件系统被设计为只能处理文本。信息的正文被限制为可打印的 ASCII 字符，不能包含任意值的字节。特别是不能直接将二进制文件作为邮件消息的正文。但人们在使用电子邮件系统的过程中发现常常需要将一些大小为几十 KB～几 MB 的小文件随同邮件一同发送，但电子邮件只能传送文本的限制导致了大大的不便。

为了适应这种需求，研究人员修改了这种模式，从而允许用电子邮件传送任意的数据（如二进制程序或图片）。一般的方案都是将数据编码为文本形式，放在邮件的消息中发送。在接收方，消息正文被抽取出来，转换回二进制形式。例如，一种方法是使用十六进制表示。二进制数据中每四位作为一个单元映射到 0～9 及 A～F 的一个字符。然后在电子邮件信息中发送这个字符序列，由接收方将这些字符翻译回二进制。

为了帮助协调和统一发送二进制数据而人们提出了多种编码方案，其中最为广泛使用的是 IETF 所提的 MIME，即多用途互联网邮件扩充（Multipurpose Internet Mail Extension）。MIME 并不指定一种二进制数据的编码标准，而是允许发送方和接收方选择方便的编码方法。在使用 MIME 时，发送方在头部包含一些附加行说明信息遵循 MIME 格式，以及在主体中增加一些附加行说明数据类型和编码。除了在发送方和接收方之间提供一致的编码方式外，MIME 还允许发送方将信息分成几个部分，并对每个部分指定不同的编码方法。这样，用户就可以在同一个信息中既发送普通文本又附加（attach）一个图像了，这就是现今邮件中的附件。当接收者查看消息时，电子邮件系统显示出文本消息，然后询问用户如何处理附加的图像（即在磁盘上保存一个副本或在屏幕上显示副本）。当用户决定了如何处理附件时，MIME 软件自动解码附加的数据。

为了透明地编码和解码，MIME 在电子邮件头部增加了两行：一行用来声明使用 MIME 生成信息，另一行说明 MIME 信息是如何包含在正文中的。例如，头部的行：

MIME-Version：　1.0

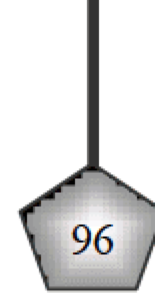
Content-Type：　Multipart/Mixed；　Boundary=Mime_separator

说明了信息是使用 MIME 版本 1.0 生成的，并且包含 Mime_separator 的行将出现在正文信息的每个部分之前。当 MIME 用来发送标准文本信息时，第二行变为：Content-Type：　text/plain。表 11.1 列出了某些公用的 MIME 内容类型。

表11.1　某些公用的MIME内容类型

内容类型	应　　用
Text/html	HTML 文档
Text/plain	纯文本文档
Image/gif	Gif 格式图像
Image/jpeg	JPEG 格式图像
Image/png	PNG 格式图像
Image/mpeg	MPEG 格式影像
Video/quicktime	Quicktime 格式影像

MIME 的主要优点在于它的灵活性。这种标准并不规定所有的发送方和接收方必须使用单一的编码方式。取而代之的是，MIME 允许使用任何时候发明的新的编码方式。发送方和接收方只要能同意一种编码方式及对该编码方式使用同一名字，就可以使用传统的电子邮件进行通信。进一步，MIME 没有规定用来划分各部分所用的具体值或用来命名编码方案的方式。发送方可以选择主体中不会出现的任意字符串作为分隔符，接收方使用头部的信息来决定怎样将信息解码。MIME 与老的电子邮件系统是兼容的。而且，传送信息的电子邮件系统不需要理解正文或 MIME 头部行所使用的编码——这些信息可以完全像任何电子邮件信息



一样对待。邮件系统传送头部信息而不解释它们，并将正文像单个文本块一样对待。

11.4 SMTP 协议

从11.2节的介绍知道邮件传送代理间通过TCP连接来传输邮件信息，但TCP连接只提供了一个可靠的信息通道，邮件传送时还必须处理许多细节。例如要允许发送方询问一个给定的邮箱在服务器所在的计算机上是否存在；要保证邮件可靠的传递——发送方必须保存一个信息的副本直到接收方将一个副本放至不易丢失的存储器（如磁盘）。这些功能不在TCP的范畴内，因此需要有专门的协议来处理这类细节。TCP/IP协议族提供了两个电子邮件传输协议：MTP（Mail Transfer Protocol，邮件传输协议）和SMTP（Simple Mail Transfer Protocol，简单邮件传输协议）。顾名思义，后者比前者简单。但简单并不意味着功能差，而正因为SMTP协议既简单又具有足够的功能，从而得到广泛的应用。目前，Internet上的绝大多数邮件服务器均使用SMTP协议，因此本节主要介绍SMTP协议的工作过程和主要命令。

11.4.1 SMTP 命令

SMTP 工作时是通过发送方发送命令，接收方处理命令后返回相应的应答；然后发送方再根据应答发送新的命令再接收应答，这样一种经过多轮命令-应答的交互来完成邮件传输的。SMTP 定义了 14 个命令，它们是：

```
HELO <SP> <domain> <CRLF>
MAIL <SP> FROM:<reverse-path> <CRLF>
RCPT <SP> TO:<forward-path> <CRLF>
DATA <CRLF>
RSET <CRLF>
SEND <SP> FROM:<reverse-path> <CRLF>
SOML <SP> FROM:<reverse-path> <CRLF>
SAML <SP> FROM:<reverse-path> <CRLF>
VRFY <SP> <string> <CRLF>
EXPN <SP> <string> <CRLF>
HELP [<SP> <string>] <CRLF>
NOOP <CRLF>
QUIT <CRLF>
TURN <CRLF>
```

其中使得 SMTP 工作的基本的命令有 8 个，分别为：HELO、MAIL、RCPT、DATA、RSET、NOOP、QUIT 和 VRFY，下面分别介绍。

（1）HELO——发件方问候收件方，后面是发件人的服务器地址或标识。收件方回答 OK 时标识自己的身份。问候和确认过程表明两台机器可以进行通信，同时状态参量被复位，缓冲区被清空。

（2）MAIL——这个命令用来开始传送邮件，它的后面跟随发件方邮件地址（返回邮件地址）。它也用来当邮件无法送达时，发送失败通知。为保证邮件的成功发送，发件方的地

址应是被对方或中间转发方同意接受的。这个命令会清空有关的缓冲区，为新的邮件做准备。

（3）RCPT——这个命令告诉收件方收件人的邮箱。当有多个收件人时，需要多次使用该命令，每次只能指明一个人。如果接收方服务器不同意转发这个地址的邮件，它必须报 550 错误代码通知发件方。如果服务器同意转发，它要更改邮件发送路径，把最开始的目的地（该服务器）换成下一个服务器。

（4）DATA——收件方把该命令之后的数据作为发送的数据。数据被加入数据缓冲区中，以单独一行是"<CRLF>.<CRLF>"的行结束数据。结束行对于接收方同时意味立即开始缓冲区内的数据传送，传送结束后清空缓冲区。如果传送接受，接收方回复 OK。

（5）REST——这个命令用来通知收件方复位，所有已存入缓冲区的收件人数据、发件人数据和待传送的数据都必须清除，接收方必须回答 OK。

（6）NOOP——这个命令不影响任何参数，只是要求接收方回答 OK，不会影响缓冲区的数据。

（7）QUIT——SMTP 要求接收方必须回答 OK，然后中断传输；在收到这个命令并回答 OK 前，收件方不得中断连接，即使传输出现错误。发件方在发出这个命令并收到 OK 答复前，也不得中断连接。

（8）VRFY——该命令使客户能够询问发送方以验证接收方地址，而无需向接收方发送邮件。通常是系统管理员在查找邮件交付差错时手工使用的。

接收方服务器处理完命令后返回的应答信息为了与命令区别开来采用了数字序列。表 11.2 给出了 SMTP 应答中用到的代码及其含义。

表11.2　SMTP应答代码

代　码	含　义
500	Syntax error, command unrecognized
501	Syntax error in parameters or arguments
502	Command not implemented
503	Bad sequence of commands
504	Command parameter not implemented
211	System status, or system help reply
214	Help message
220	<domain> Service ready
221	<domain> Service closing transmission channel
421	<domain> Service not available, closing transmission channel
250	Requested mail action okay, completed
251	User not local; will forward to <forward-path>
450	Requested mail action not taken: mailbox unavailable
550	Requested action not taken: mailbox unavailable
451	Requested action aborted: error in processing
551	User not local; please try <forward-path>
452	Requested action not taken: insufficient system storage
552	Requested mail action aborted: exceeded storage allocation

续表

代　码	含　义
553	Requested action not taken: mailbox name not allowed
354	Start mail input; end with <CRLF>.<CRLF>
554	Transaction failed

11.4.2 SMTP 工作过程

下面以 RFC821 中的一个例子来进一步说明 SMTP 协议的工作过程。

假设在 Alpha.ARPA 主机的 Smith 发送邮件给 Beta.ARPA 主机的 Jones，Green 和 Brown 的，这里假定主机 Alpha 与主机 Beta 直接相连。此时，SMTP 协议的工作过程如图 11.3 所示。

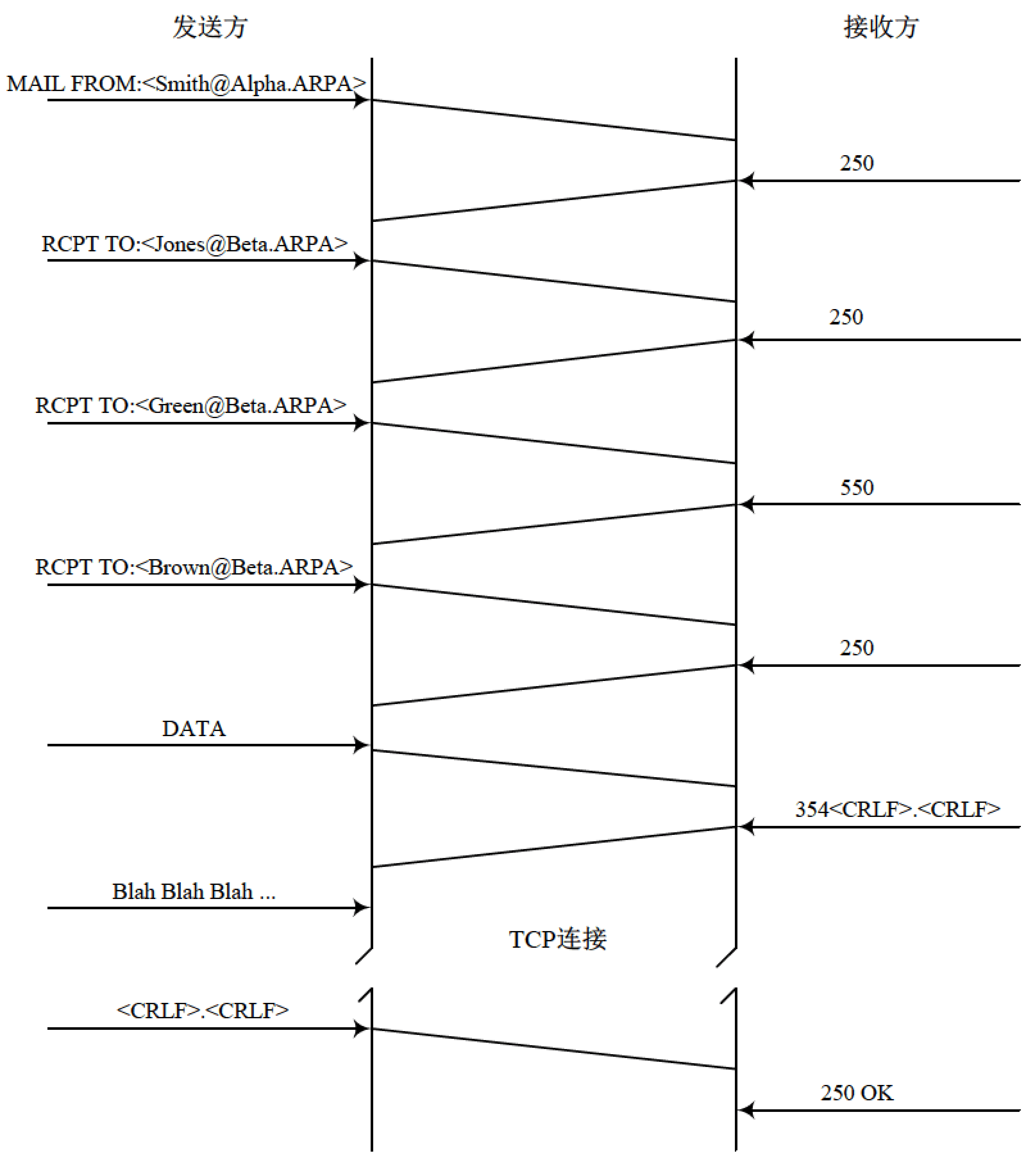


图 11.3　SMTP 工作过程示例

发送方在邮件发送时首先发送 Mail 命令通知接收方邮件发送者的地址，在获取肯定答复后（应答 250），发送 RCPT 命令告知接收方收件人地址；在本例中收件人 Jones 和 Brown

存在而收件人 Green 不在主机 Beta 上（回复 550），当发送方得知有存在的收件人后将发送 DATA 命令请求传送数据，对此命令的肯定答复是 354 并以<CRLF>.<CRLF>结束；然后开始邮件的传输。传输结束后发送方发送<CRLF>.<CRLF>以通知接收方邮件传送结束。最终，Jones 和 Brown 将正确接收邮件。

11.5 邮 箱 访 问

从 11.2 节的 TCP/IP 邮件系统结构可知，发送方的传送代理并不直接对接收者邮箱进行存取，而是每个有邮箱的计算机系统必须运行一个邮件服务器程序来接收电子邮件并将它放进正确的邮箱。因此邮箱通常处于邮件服务器所在的机器上。若在用户本地计算机上放置邮箱并运行邮件服务器会存在下述一些问题。首先，一般用户使用的 PC 机计算存储资源有限，可能难以支撑邮件服务器的运行。其次，用户的计算机一般没有固定的 IP 地址，即使有也不能保证全天 24 小时的开机，显然一台在一些时间里（如工作时间之外）处于关闭状态或是不与因特网相连的计算机不足以作为电子邮件接收方。再者，即使用户的机器上述条件都满足，将邮箱放在用户本地机还存在如下不便。即用户可能需要在多个地点访问邮箱，比如将邮箱放置在用户单位的机器上，那么他在家里时将难以访问邮箱。因此用户一般使用的邮箱是由因特网服务提供商或用户所在单位部门提供的，这些邮件服务器并不在用户的本地计算机上运行。这就需要解决用户如何访问邮箱的问题。

TCP/IP 协议包含了提供对电子邮件邮箱进行远程存取的协议。这些协议允许用户的邮箱安置于运行邮件服务器的计算机上，并允许用户从另一台计算机对邮箱的内容进行存取。其中使用最为广泛的是邮局协议（Post Office Protocol，简称 POP），目前使用的大多是邮局协议第三版，即 POP3。

11.5.1　POP3 协议

POP3 协议在工作时采用客户机/服务器模式。需要在邮箱所在的机器上运行 POP 服务器，用户运行的电子邮件软件成为该 POP 服务器的客户，对邮箱的内容进行存取。图 11.4 描述 POP 协议的工作模式。

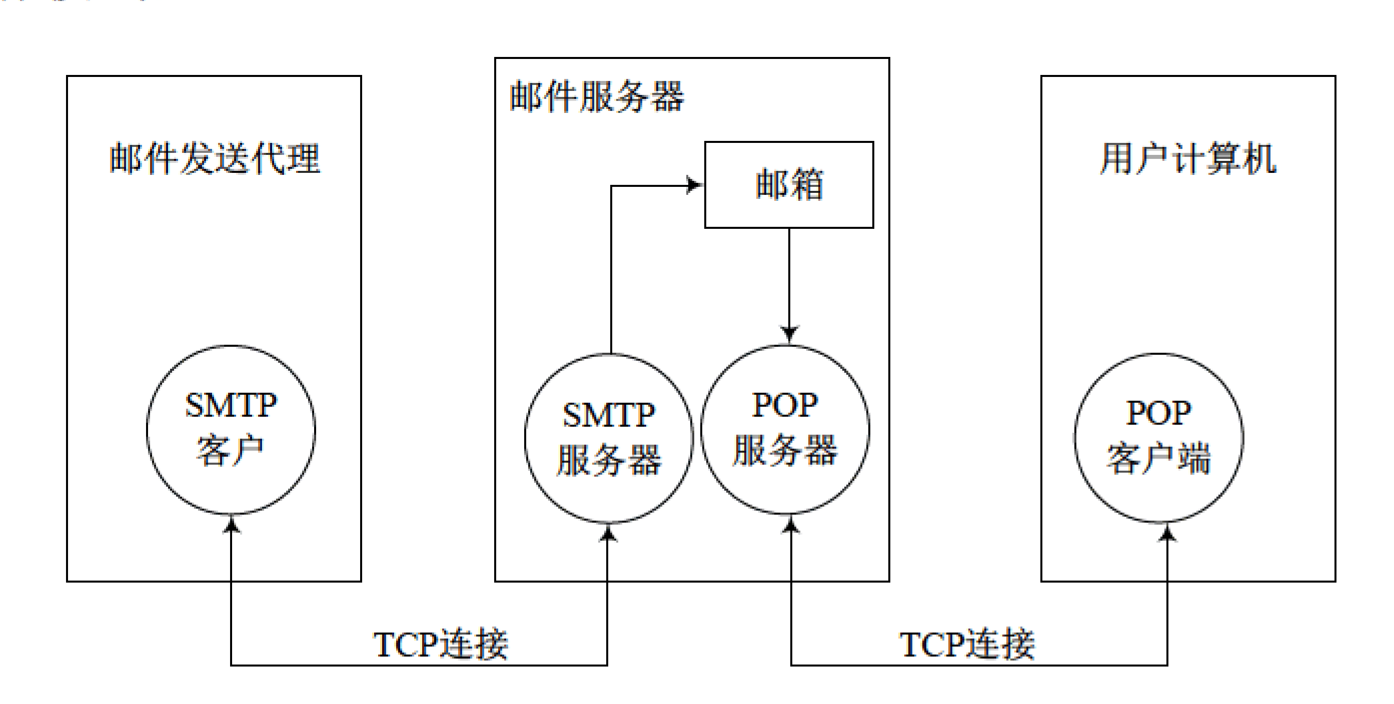


图 11.4　POP 协议工作模式



邮件发送方利用邮件传送代理与邮件服务器上的 SMTP 服务器建立连接，将邮件放入用户邮箱。邮件服务器上的 POP 服务器侦听 TCP 的 110 端口，用户在访问邮箱时用户的邮件软件作为客户端与 POP 服务器建立 TCP 连接，然后通过交互一系列的命令将用户邮箱中的邮件下载到用户本地计算机或将邮件删除。这里命令的交互过程与 SMTP 的命令交互过程类似，故不再赘述。

11.5.2 其他邮箱访问方式

在使用 POP 协议的访问邮箱时，所有信息都下载到客户端，通常是与服务器接通即下载。随后，所有进程包括读取、删除、存储，仅在客户端进行。这种访问属于脱机式的访问。当然，信息也可用脱机方式创建，之后通过 SMTP 上载到服务器。目前 Internet 上的邮箱访问除了脱机方式外还有联机方式和分离式访问。

所谓联机方式是通过将所有信息回溯至主机和外部账号进行处理。这里，客户端用户在服务器上操作邮箱数据，通过会话保持链接。客户端在本地不保存任何邮件信息，只是在需要时才从服务器检索。联机方式的邮箱访问一般采用 HTTP 协议，通常浏览器作为邮箱访问的客户端，这种方式又称作 webmail。因此，若不连接到网络，就不能对邮件实施任何操作。

分离式访问是联机模式与脱机模式的混合方式。在这种模式下，终端用户可以周期性地链接到服务器，如在家中通过拨号接入，下载信息。信息可以在脱机状态读取、删除或重组。若下一次链接到来，则将服务器中刚才正被访问的远程信息实施同步存储。其结果是，分离式访问用户可以像联机方式那样进行存取。分离式访问采用 IMAP 协议（Internet Message Access Protocol），IMAP 提供的摘要浏览功能可以让用户在阅读完所有的邮件到达时间、主题、发件人、大小等信息后才作出是否下载的决定。也就是说，用户不必等所有的邮件都下载完毕后才知道究竟邮件里都有些什么。同时 IMAP 允许用户在服务器上建立任意层次结构的文件夹，并且可以灵活地在文件夹之间移动邮件，随心所欲地组织邮箱（这些显然是通过 POP3 做不到的）。

从三种访问方式的功能来看，分离式访问同时具有 POP 方式和 webmail 的优点，功能最为强大，使用方便灵活。但该种方式占用服务器资源较多，需要较多的技术支撑，因此采用该种方式的运营商较少。POP 方式因其简单且使用也较为方便而得到广泛的使用。

第 12 章　HTTP 协议

最初设想 Internet 时，设计者的着眼点主要在于通信的安全性和可靠性。在第一批连接建立之后不久，人们很快就发现，Internet 可方便地用于信息交流和项目协作。不过即使如此，创建之初的 Internet 用量增长缓慢，主要停留在较大型的单位和大学中。造成这一局面的主要原因是 Internet 上最初开发出的 Telnet、FTP 等应用本身不适用于信息的发布和交流。这一局面随着 WWW（World Wide Web）技术的出现发生了根本地改观。尤其是 1993 年以后图形 WWW 浏览器的开发成功直接导致 Internet 用户和节点的指数增长。图 12.1 显示了 1994 年 1 月至 1995 年 3 月份间作为 WWW 基础的 HTTP 协议分组在整个 Internet 分组中所占比例的增长情况。WWW 之所以能受到广泛的欢迎是因为 WWW 采用了图形用户界面，在 Web 页面中融合了文本、图像、声音和视频等多种信息表达方式，同时其中的超链接允许用户只需简单的单击操作即可从一个网站跳转至另一个网站，这不仅极大方便了信息的发布也大大简化了用户对信息的浏览和检索。

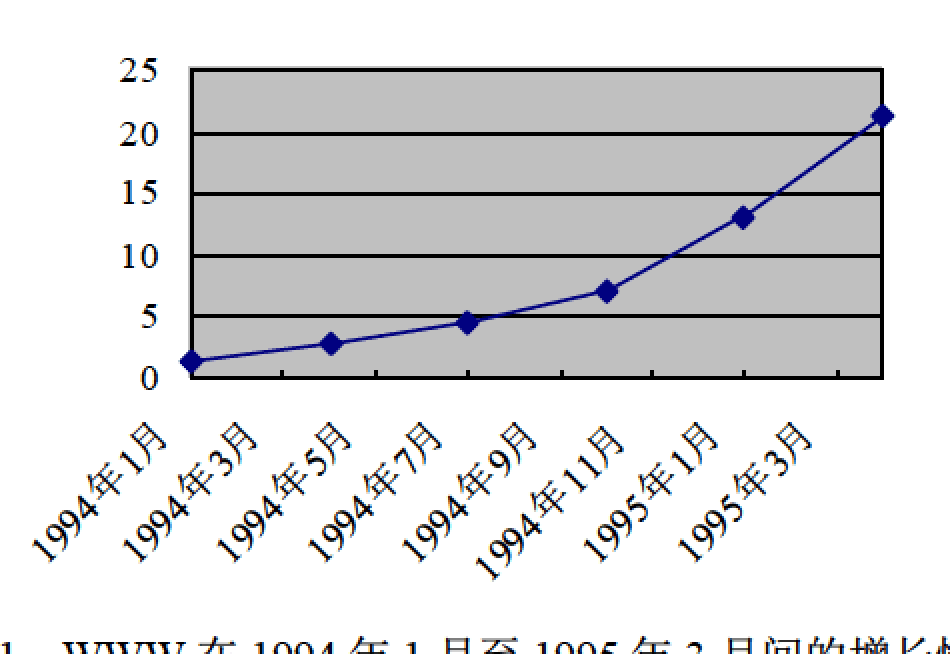


图 12.1　WWW 在 1994 年 1 月至 1995 年 3 月间的增长情况

本章着重介绍构成 WWW 基础的 HTTP 协议，谈到 HTTP 协议必然涉及到 HTML 语言，因此首先简要介绍 HTML 语言，然后讨论 HTTP 协议的工作模式、主要命令以及 HTTP 协议的工作过程。

12.1　超文本和 URL

12.1.1　超文本

WWW 作为 Internet 上信息发布的场所其本质上可看作是信息的集合。而这些信息是以文档形式分布存放在 Internet 中的主机上。那么如何在一个文档中组织这些分散存放的信息呢？超文本概念在这里发挥了很大作用。所谓超文本文档是指该文档中除了基本的信息外，



文档中还可以有指向信息集合中其他文档的指针。超文本概念提供了一种分布环境下组织数据和信息的机制。其实超文本概念并不是 Internet 时代才出现的，作为一种组织信息的手段在 20 世纪 60 年代即由斯坦福研究所的 Ted Nelson 提出。CERN 小组继续了这一思想在 1990 年提出了超文本标记语言 HTML（HyperText Marked Language），并将它用于 Internet 上形成最初的 WWW。不过这时的超文本几乎是名副其实的超文本，是在文本文档的基础上增加了指向其他文本文档的指针，并未引起广泛的注意，1990 年至 1992 年之间 WWW 增长还很缓慢。随着 1993 年支持图形图像和声音等信息的浏览器开发成功 WWW 迎来了空前的发展。此时的超文本文档中不仅包含文本信息还包含了图形、图像和声音视频等指针，实际上已发展为超媒体，不过习惯上仍将其称为超文本。

12.1.2　统一资源定位 URL

超文本中一个关键的概念是指向其他文档的指针，由于 WWW 是分布式的，并且指针指向的文档类型多样，这给如何描述这类指针带来了一定的困难。首先，指针必须标明所指向文档的存放地点和文档名；其次由于 Internet 上有各种各样的应用，并且将来还可能出现一些新的应用，指针中应有采用哪种应用来获取所指向的文档，并且这种方法应对将来出现的新应用同样有效。为此人们发明了一种语法格式，用来组织描述远程项的各种信息。该语法把信息编码成一个字符串，称为统一资源定位（Uniform Resource Locator，URL）。URL 的一般形式为：

protocol：//computer_name：port/document_name

URL 被冒号和斜杠符分隔成三个部分：协议，计算机名与文档名。其中 protocol 是访问文档所采用应用所用的协议名，computer_name 是文档所在计算机的域名；计算机名后的冒号表示 port——协议端口号是可选项（ 由于可选端口号很少使用， 在以下的讨论中将忽略此项）；document_name 是在指定计算机上的文档名，包括路径和文件名。例如，URL：

http:/ / www.seu.edu.cn/cs/index.html

指明协议为 http，计算机名为 www.seu.edu.cn， 文件 cs/ index.html。

12.2　HTML 简介

12.2.1　超文本文档结构

超文本文档是 ASCII 字符文件，但不同于一般的文本文件，它是格式化的文件。其格式是由 HTML 语言来定义描述的。HTML 语言定义了超文本文档的结构，描述了文档各部分在浏览器中处理时的相关信息（但不指定浏览器如何格式化文档）。HTML 语言是通过定义一系列的标签来实现这些功能的。

每个 HTML 文档分为两个主要部分：头部后紧跟着主体。头部包含了文档的细节，而主体则包含了大部分信息。例如，头部包含了文档的标题——大多数的浏览器用标题作为标签来让用户知道哪一网页正在被浏览。在语句构成上，每个 HTML 文档以一个包含标签和其他

信息的文本文件来表示。在大多数的编程语言中，可以在文档中插入空白字符（例如空行与空格字符）增加源程序的可读性，那些空白字符对于浏览器所显示的格式版本毫无影响。HTML 标签为文档提供结构提示和格式提示。一些标签指定一个立即生效的动作（例如在显示屏上移动到一个新行），标签被置于动作应该出现的地方。其他的标签被用于指定一个适用于紧跟在标签之后的所有文本的格式操作。这些标签成双出现，其中开始标签和结束标签分别启动和结束动作。标签被用于指定一个立即动作或者启动一个以小于和大于符号括起来的标签名形式出现的格式动作（尽管标签名不区分大小写，但是按照惯例以大写来表示）。

```
<TAGNAME>
```

相应的用于结束的标签以两个连续的小于符和斜杠符开始，并且以一个大于符结束：

```
</TAGNAME>
```

例如，HTML 文档以标签</HTML>开始。标签<HEAD>与</ HEAD >包括了头部，而标签<BODY >与</BODY>包括了主体部分。在头部，标签<TITLE>与</TITLE>包括了形成标题的文本。图 12.2 举例说明了 HTML 文档的一般形式。

```
<HTML>
    <HEAD>
        <TITLE>
            Hello,World!
        </TITLE>
    </HEAD>

    <BODY>
        Hello,World!
    </BODY>
</HTML>
```

图 12.2　HTML 文档的一般形式

图中每个标签都出现在一个新行上，并且以行首缩进来显示结构。不过，这种惯例只是为了方便对文档的阅读，就如采用缩进格式缩写的 C 语言程序一样，其中的空格不会影响页面在浏览器中的显示。这也是 HTML 文档和文本文档不同的地方，HTML 文档的显示格式是由文档中的标签定义的。

12.2.2　HTML 中常用标签

从上面的介绍可知，一般标签是成对出现的，将<tag>和</tag>的组合再加上它们中间的内容称为网页的元素。这些元素在浏览器中的处理是由其中的标签来决定的。下面将介绍 HTML 文档中常见的标签。

1．<HTML>和</HTML>标签

这对标签用来表示 HTML 文档的开始和结束，其作用是通知浏览器它所处理的是 HTML 文档。实际上，可以把 HTML 格式看作是一个有卷心菜结构的文档。也就是说它具有一个连续的表层。其最外层即是以<HTML>和</HTML>标记的，所有其他的元素均包含在这个主要的 HTML 元素内部。



2．<HEAD>和</HEAD>标签

一个 HTML 文档由头部和主体构成。文档的头部是由标签<HEAD>和</HEAD>标出。头部一般包含了文档的标题和索引等文档的背景信息。

3．HEAD 元素

在文档<HEAD>层内的网页元素称为 HEAD 元素。HEAD 元素主要用以标识文档的题目和一些索引之类的背景信息。Web 浏览器并不显示 HEAD 元素的内容，所以在浏览 Web 页时是看不到的。表 12.1 列出了基本的 HEAD 元素。

表12.1　基本的HEAD元素

元　　素	描　　述
TITLE	用于 Web 页的跟踪/访问
BASE	在 Web 页中标识文档的 URL 地址（统一资源定位）
ISINDEX	通知 Web 浏览器该文档可被搜寻
LINK	描述文档和其他文档之间的链接
*Href	标识其他的文档链接
*Name	为链接取名
*Rel	描述与其他文档的关系
*Rev	描述与其他文档的关系
*Urn	统一资源名称
*Methods	其他文档所支持的 HTTP 方法
META	Meta 信息
*Http-*quiv	将 META 元素与一个协议连接起来
*Name	将内容命名
*Content	将文档中的信息分类
NEXTID	识别代码
*N	定义下一个识别代码

（1）TITLE

TITLE 元素可以帮助用户表示和跟踪 Web 页。通常会在 Web 浏览器的标题栏（但不在 Web 页本身）中显示定义的标题。它必须少于 50 个字符，并且不能包含其他 HTML 元素或属性。格式为：

```
<TITLE>字符串</TITLE>
```

（2）BASE

BASE 元素标识了在 Web 页中使用的其他文档的 URL 地址。如果一个文档被移开了它原来所在的位置，就可以指明它最初的来源。这将使得超文本链接的装入更加精确。BASE 元素有一个属性，就是用来标识其他资源的 URL 的 Href。例如：

```
<BASE Href="http//www.sina.com">
```

（3）ISINDEX

ISINDEX 元素用来通知 Web 浏览器在 BASE 元素中列出的文档是可被搜寻的，例如，

将该标记和 HEAD 标记联合起来使用便可允许搜寻整个文档，该文件所在的服务器必须能够支持搜寻。

（4）LINK

LINK 元素给出了当前的文档和其他文档或对象之间的关系的详细描述。例如：

```
<LINK Href="my_info.htm">
```

LINK 元素有下述一些属性：

- Href：给出了该连接所描述的文档的名字。
- Name：将链接命名以使它可以作为一个可能的超文本目标来使用。
- Rel：描述由链接所定义的关系。例如，Rel=“made”的意思是，在 Href 中给定的 URL 是文档的作者。
- Rev：与 Rel 描述的关系恰好相反。例如，Rev=“made”的意思是，当前的文档是 Href 中所给定的 URL 的作者。
- Urn：表明该文档的 Uniform Resource Name（统一资源名称）。

4．<BODY>和</BODY>标签

<BODY>和</BODY>标签中所包含的文档的主体部分。BODY 中的元素是 Web 浏览器将要进行显示的那一部分内容，即 BODY 中元素将影响文档的外观和式样。BODY 部分可以包含文字、图形图像、表格等元素。

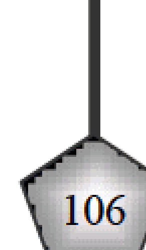
由于 HTML 文档存储在文本文件中，所以文档必须包含明确的标签来说明输出是如何显示的。例如：
标签指导浏览器引入一个行分隔符。也就是说，当在输入中遇上
时，浏览器在产生更多的输出前在显示屏上移动到下一行的行首。<p>和</p>标签用以标记一个段落元素。它允许在 Web 页中创建文字块。而<LI>是一个列表元素。它标记着列表中每个项目的开始。

Web 文档中还可以包含非文本信息。通常，非文本的信息诸如图形或者数字相片等并不直接插入于文档之中。数据位于一个独立的地点，而文档包含了指向数据的引用。当浏览器遇上这些引用时，浏览器去指定地点取得图像，并且将图像插入到所显示的文档中。例如，<IMG>标签用来标记 HTML 文档引用的外部图像。如，<IMG Src=“PHOTO.jpg”>表明文件“PHOTO.jpg”包含一个浏览器所要插入到文档中的图像。其中 IMG 的 Src 属性指明了图像的来源。

除了上面简单介绍的这些标签外，HTML 语言还定义足够的标签用于显示格式的控制，文档显示的背景控制等，这里不再一一介绍，有兴趣的读者可以参阅 HTML 方面的相关文献。

12.3　HTTP 协议概述

超文本传送协议（HyperText Transfer Protocol，HTTP 协议）是 Web 服务器用来处理服务器和客户机之间的数据流的协议。HTTP 协议和 HTML 语言构成了 WWW 的技术基础。HTTP 是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于 1990 年提出，经过几年的使用与发展，得到不断地完善和扩展。目前在 WWW 中使用的是 HTTP/1.0 的第 6 版，HTTP/1.1 的规范化工作正在进行之中，而且 HTTP-NG（Next Generation of HTTP）的建议已经提出。



12.3.1　HTTP 协议的工作模式

HTTP 协议是一个简单的协议，与其他 Internet 上的应用协议类似，HTTP 协议是基于请求/响应模式的。为此 HTTP 协议定义了一组消息，这些消息分为两种类型：来自客户机的“请求”消息和来自服务器的“应答”消息。HTTP 协议在工作时 Web 浏览器通常充当客户端的角色，当用户向浏览器提交命令后，浏览器将打开与远端服务器 TCP 连接的 80 端口（80 端口是 HTTP 协议的默认端口，当然采用其他端口的 HTTP 服务器也是存在的），然后在此连接上发送相应的请求命令。服务器在收到请求命令对其做出相应处理后将处理的结果以应答消息返回到客户端并关闭此次 TCP 连接。其工作过程如图 12.3 所示。

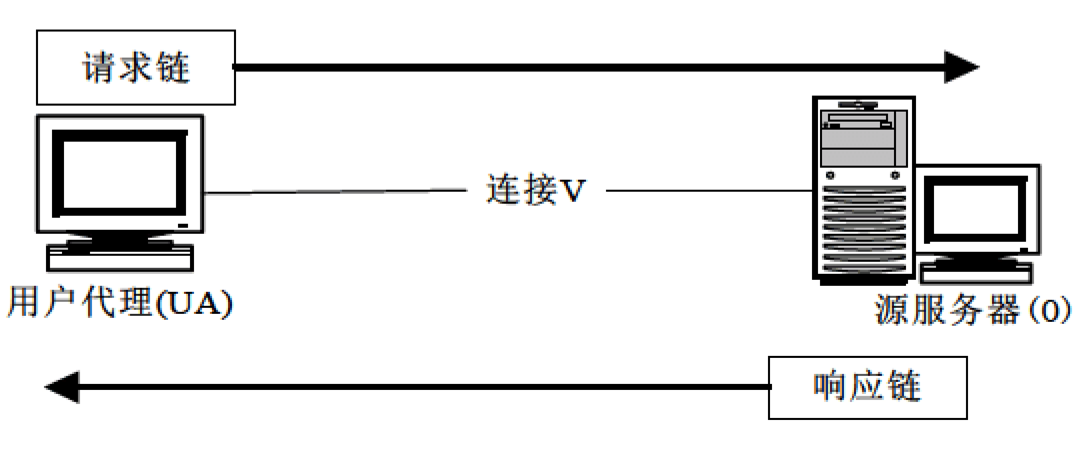


图 12.3　HTTP 协议的工作过程

12.3.2　HTTP 协议特点

从上述 HTTP 协议的工作模式可以看出 HTTP 协议具有如下主要特点：

（1）支持客户/服务器模式。

（2）简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。

（3）灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。

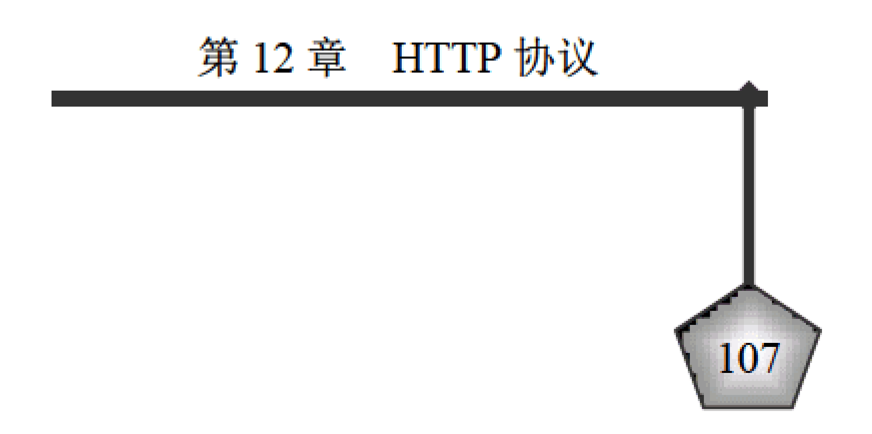
（4）无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

（5）无状态：HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

12.4　HTTP 请求和应答

12.4.1　请求消息

请求消息由客户机发送给服务器以请求数据。典型的 HTTP 请求消息格式如下：




```
request-line (<method> URI <HTTP-version>)
   headers
   <blank line>
   <body>
```

其中第一行是请求行，method 是客户发送的具体的请求方法，URI（Uniform Resource Identifier，URI）是所请求页面的统一资源标识，最后部分是客户端所用的 HTTP 协议的版本信息。请求行和随后的首部信息（headers）构成请求消息的报文头，报文头和报文主体以空行分割开。首部信息可以包含 0 个或多个字段。

HTTP/1.0 支持三个请求方法：

（1）GET 请求，返回请求行中 URI 所指出的页面信息。

（2）HEAD 请求，类似于 GET 请求，但服务器程序只返回 URI 指定文档的首部信息，而不包含实际的文档内容。该请求通常被用来测试超文本链接的正确性、可访问性和最近的修改。GET 和 HEAD 请求报文不带报文主体。

（3）POST 请求用来发送电子邮件、新闻或发送能由交互用户填写的表格。这是惟一需要在请求中发送 body 的请求。使用 POST 请求时需要在报文首部 Content-Length 字段中指出 body 的长度。

12.4.2　应答消息

服务器以如下方式向客户机发送一个应答：

```
status-line (<HTTP-version> response-code response-phrase)
headers
<blank line>
<body>
```

应答消息的格式与请求消息类似，分为消息头和主体两部分，中间以一空白行隔开。应答的第一部分是应答头。它以一个状态行开始，状态行包括所用的 HTTP 版本、一个状态编码（表 12.2 给出各种状态编码及其含义）和一个原因短语。跟随在状态行之后的是描述应答细节的一系列格式化首部字段。跟随在应答头后的空白行说明应答头已结束。如果有与应答有关的数据体，它将跟随在空白行后，即上面的 body 部分。

表12.2　HTTP应答状态编码

状　态　码	原因短语
信息（Informational）1xx HTTP/1.0 中为定义	
成功（Successful）2xx	
200	正确（OK）
201	创建（Created）
202	接收（Accepted）
204	无内容（No Content）

续表

状　态　码	原因短语
重定向（Redirection）3xx	
300	多种选择（Multiple Choices）
301	永久移动（Moved Permanently）
302	暂时移动（Moved Temporarily）
304	未被修改（Not Modified）
客户机错误（Client Error）4xx	
400	错误请求（Bad Choice）
401	未授权（Unauthorized）
403	禁止（Forbidden）
404	未发现（Not Found）
服务器错误（Server Error）5xx	
500	内部服务器错误（Internal Server Error）
501	未实现（Not Implemented）
502	错误网关（Bad Gateway）
503	服务未提供（Service Unavailable）

12.4.3　首部字段

首部字段又称为元信息，即关于信息的信息。利用元信息可以实现有条件的请求或应答。请求消息中的首部字段告诉服务器怎样解释本次请求，主要包括用户可以接受的数据类型、压缩方法和语言等。而应答消息中的首部字段主要包括实体信息类型、长度、压缩方法、最后一次修改时间、数据有效期等。常见的首部字段如表 12.3 所示。

表12.3　常见首部字段表

首部名称	含　　义	请　求	应　答	主　体
Allow	主体所允许的方法			●
Authorization	客户授权信息	●		
Content-Encoding	主体所用的编码			●
Content-Length	主体长度			●
Content-Type	主体类型			●
Date	客户或服务器的时间	●	●	
Expires	主体的有效期			●
From	请求发送者的 E-mail	●		
If-Modified-Since	某时间前网页是否更改	●		
Last-Modified	最后更改的时间			●
Location	请求页重定向后的位置		●	
Pragma	客户机或服务器实现细节	●	●	
Referer	客户获取所请求 URI 的 URI	●		
Server	服务器信息		●	

续表

首部名称	含　　义	请　求	应　答	主　体
User-Agent	客户机信息	●		
WWW-Authenticate	服务器要求授权信息		●	
Access Authentication	授权信息	●	●	

一个首部字段由字段名和随后的冒号、一个空格和字段值组成，字段名不区分大小写。首部字段可分为三类：一类应用于请求，一类应用于响应，还有一类描述主体。有一些报文头（例如：Date）既可用于请求又可用于响应。描述主体的首部字段可以出现在 POST 请求和所有响应报文中。

12.5　浏　览　器

WWW 是一个分布式的超媒体系统，在 WWW 上发布的信息有文本、图像、图形、音频和视频等各种格式。Web 浏览器作为 WWW 的客户应用程序不仅要能完成 HTTP 的通信，更重要的是要能处理各种格式的信息。事实上如前所述，HTML 语言和 HTTP 协议在 1990 年就已出现，但直到 1993 年图形化的 Web 浏览器出现后 WWW 才迎来高速地增长。

Web 浏览器具有一个比 Web 服务器更为复杂的结构。服务器重复地执行一个简单的任务：等待浏览器打开一个连接并且请求一个指定的网页。随后服务器发送所请求的项，关闭连接并且等待下一次的连接。浏览器则需要处理文档的细节并进行显示。浏览器包含几个大型的软件组件，它们一起工作从而提供一个无缝服务。图 12.4 说明了浏览器概念上的组织。

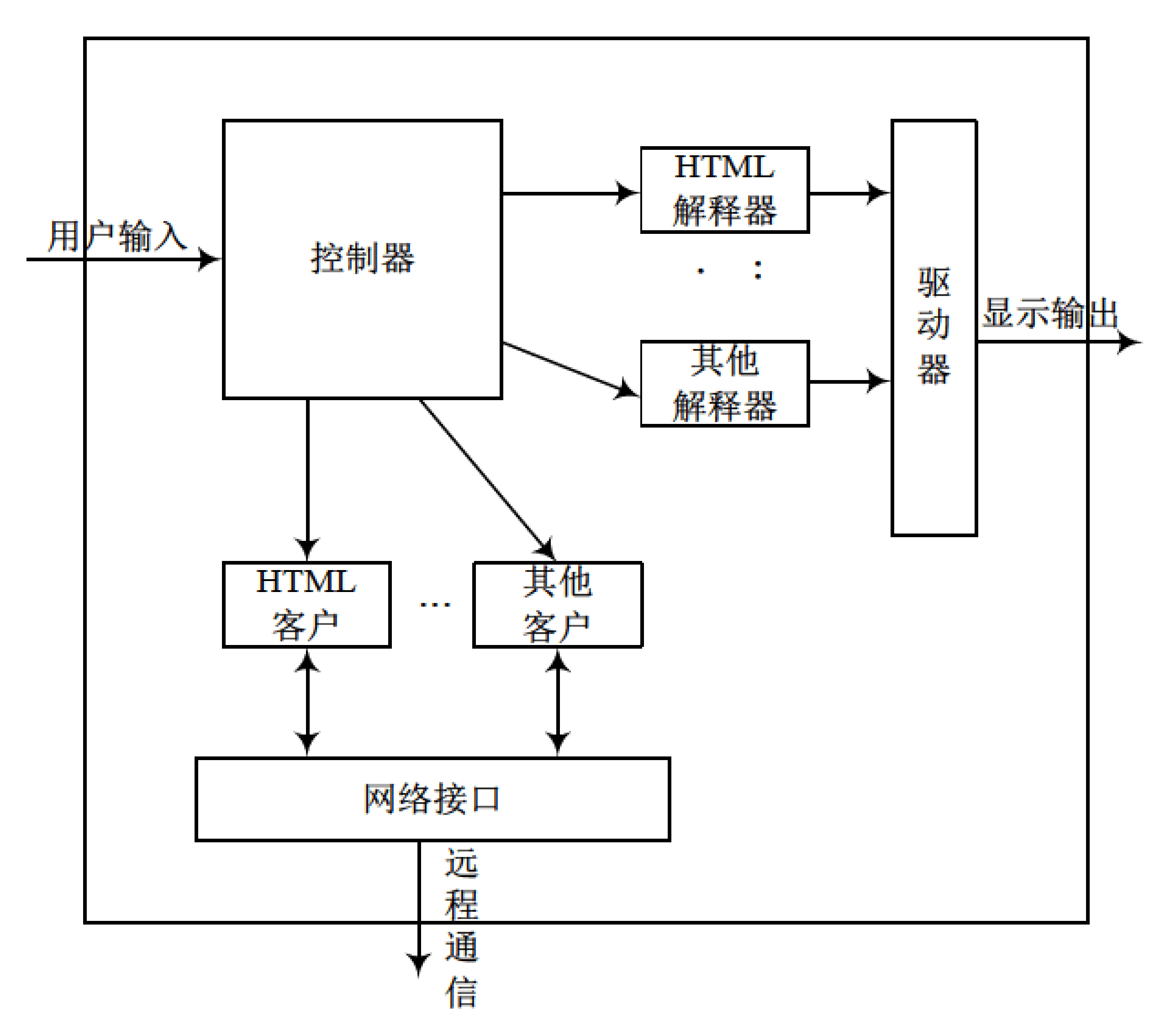


图 12.4　浏览器组织结构图

从概念上讲，浏览器由一组客户、一组解释器和一个管理它们的控制器所组成。控制器



形成了浏览器的中心部件，它控制鼠标单击与键盘输入，并且调用其他组件来执行用户指定的操作。例如，当用户输入一个 URL 或者单击一个超文本引用时，控制器调用一个客户从所需文档所在的远程服务器上取回该文档，并且调用解释器向用户显示该文档。

每个解释器必须包含一个 HTML 解释器来显示文档。其他的解释器是可选的。HTML 解释器的输入由符合 HTML 语法的文档所组成，输出由位于用户显示器上的格式化文档组成。解释器通过将 HTML 规格转换成适合用户显示硬件的命令来处理版面细节。例如，如果碰到文档的头部标签，解释器则改变用于显示头部的文本大小。同样，如果碰到一个断行标签，解释器则输出一个新行。

HTML 解释器一个最重要的功能是可包含可选项。解释器必须存储关于显示器上位置之间关系的信息和 HTML 文档中被锚定的项。当用户用鼠标选择了一个项时，浏览器通过当前的光标位置和存储的位置信息来决定哪个项被用户选中。

浏览器这种将控制器、客户组件和解释器分开的设计提供了很好的灵活性。WWW 作为超媒体系统包含了各种各样的媒体，同时随着技术的进步新技术新应用会不断地涌现，浏览器的这种面向对象的设计可以方便的通过增加插件的方式升级现有的浏览器，同时也给用户以定制浏览器功能的余地。

第 13 章　网络文件

TCP/IP 的一个重要应用就是用于不同主机系统间的数据（文件）共享。访问远程数据的方法主要有两种形式：整体文件复制和实时在线共享。常见的用于文件共享的应用协议有三种：文件传输协议 FTP 是 TCP/IP 协议族中主要的文件传输协议，它使用整体文件复制向用户提供了浏览远程文件服务器文件目录以及双向传送文件的能力。轻量级文件传输协议 TFTP 向那些只需要文件传输的应用提供了一种简单小巧的 FTP 替换方案。因为它可以小到存放在 ROM 里，TFTP 可以被用来启动无驱主机。网络文件系统 NFS 是由 SUN 微系统公司设计用来提供实时在线文件共享的应用协议，它使用 UDP 来进行消息传送并实现 SUN 的远程过程调用（RPC）机制和扩展数据格式表示（XDR）机制。因为 RPC 和 XDR 是和 NFS 独立定义的，程序员可以使用它们来构建分布式应用程序。本章分别介绍这三种协议。

13.1　FTP 文件传输协议

13.1.1　简介

FTP 协议位于 OSI 网络七层模型的应用层，同时也是 TCP/IP 协议族的一部分。本节将主要介绍 FTP 协议的设计思想、工作原理并提供一个典型的用户交互的样例加以说明。读者在通读本章后将会发现 FTP 协议作为一个广泛使用的文件传输协议其实是构建在 TCP 协议和 Telnet 协议之上的。

FTP 协议的原始设计目标有 4 个：（1）促进文件（包括程序和数据）的共享；（2）鼓励间接地或隐式地（通过程序）来使用远程计算机；（3）使得不同主机的不同文件存储系统对用户来讲是透明的；（4）高效可靠地传输数据。尽管 FTP 协议可由用户直接在客户端使用，但其主要是针对应用程序进行设计的。

13.1.2　文件访问和传输

许多网络系统都提供了访问远程主机上文件的能力。设计者尝试了各种不同的方法来实现远程文件访问，每种方法有其各自的目标。例如为了降低计算机系统的整体成本，设计者使用一个中心文件服务器来为没有本地磁盘的设备提供文件存储。这些无盘设备可以是移动的手持设备，可以通过高速无线网络访问文件服务器。另一个例子是使用远程文件访问来备份数据，本地的计算机存储系统周期性地把本地的一些文件传输到远程计算机上以防备重要数据丢失。而在另一些情况下，设计者更关心的是数据能够在多个应用程序、多个用户及多个主机之间进行共享，一个组织可能会采用一个独立的对数据进行规范管理的数据库服务器



来为组织内的各个用户提供文件共享服务。

13.1.3　在线共享访问

文件共享主要有两种方式：在线及时共享和文件整体复制。在线共享访问意味着多个应用程序可以同时访问同一个文件，对文件的改动将随即影响所有访问它的程序。而文件整体复制是指当程序需要访问一个文件时，它将获得该文件的一个本地复制，复制通常是用于只读的数据，但一旦数据需要被修改，它必须修改本地的这份复制并覆盖远程原始主机的该文件。

许多人认为在线共享访问只能通过数据库服务器的方式来提供文件的共享访问，但实际上文件共享并没有这么复杂化并且是相当易于使用的。在线文件共享并不需要远程客户端像数据库系统那样使用一个特定的客户端软件，而是把远程文件系统集成到本地文件系统中来的，对于用户来讲这种集成是透明的，可以像使用本地文件一样来使用远程文件。可以把远程文件作为应用程序的输入和输出。

远程文件共享透明化的好处是显而易见的，因为对应用程序来讲远程文件和本地文件是无区别的。用户程序就可以同时访问本地和远程文件并对它们中的数据进行互操作。相对而言，这样做的缺点则不是很明显，其缺点在于如果在程序运行过程当中网络或者远程主机出现故障，则应用程序将无法正常运行。即便远程主机不出现故障，但是网络或者主机的负载过重时，也会使得程序运行得很缓慢，甚至超时出错，并且这样的状况出现时不可预料，最终结果将会使得应用程序的稳定性大大降低。

姑且忽视这些缺点，要实现一个集成的透明的文件共享系统也是非常困难的。在一个异构的系统中，一台机器上的文件名未必都能转换成另一台机器中的文件名。相应的，一个远程文件共享系统还必须要解决文件所有权、用户授权和访问保护等问题，而这些问题是不能够跨主机传递的。最后对各个文件的表示和操作各个系统也是不一样的，要对所有的文件实现所有的操作是很难的甚至是不可能的。

13.1.4　文件传输共享

对实时在线文件共享方式的替换手段就是文件传输共享。使用文件传输机制的远程数据访问可分为两步：首先用户获得远程文件的一份本地复制，然后再对这份复制进行操作。大多数的文件传输共享机制是和本地的文件系统相分离的，用户必须通过调用一个特定的客户端软件来获取远程文件的复制。调用客户端软件时，用户首先指明远程文件所在的机器地址，并且可能要提供一定的认证信息（如用户名、口令）来获取文件的访问权，一旦文件传输结束，用户结束客户端程序，就可以使用本地的应用程序来处理这份复制了。整体文件复制的一个优点是一旦传输完毕，本机系统就可以对它进行完全的有效的访问和操作，所以在通常情况下这种方式要比实时在线文件共享程序运行效率更高。

和在线系统一样，在异构系统间的整体文件传输可能是很困难的。客户端和服务器必须要在用户授权、文件所有权、访问保护和数据格式等问题上达成一致，后者显得尤为重要，因为处理得不好的话逆向传输将无法实现。文件表示的具体格式差异以及为了消除这些差异所采用的技术和文件传输中所涉及的具体系统紧密相关。进一步地说，我们现在还不能解决

所有异构系统的数据表示问题，从一种格式转化为另一种格式可能会造成数据丢失。但是倒并不需要了解所有系统间的具体差异，这些差异也不是至关重要的，记住 TCP/IP 协议族本身就是针对异构系统间的交互而设计的，FTP 协议作为其中的一部分将会体现这样的特点。

13.1.5 FTP 协议的特点

文件传输是使用最广泛的 TCP/IP 应用之一，在现有的网络流量中占相当大的比例。其实在 TCP/IP 成为实际可用的标准之前，标准的文件传输协议就已经存在了，这些早期的文件传输标准已被集成到现在的这个文件传输协议（File Transfer Protocol）中，主要以 RFC959 为规范。

如果已有一个可靠的端到端的传输协议如 TCP，那么文件传输看起来也许是很容易的事情。不过，事实上正如在前文中所指出的，在异构系统间的授权、命名、表示等工作使得这个协议变得很复杂。也就是说 FTP 协议还提供了除文件传输本身之外的其他一些功能。

1．交互式访问

尽管 FTP 协议是针对应用程序的使用所设计的，但大多数的协议实现都为用户自己与远程服务器的交互提供了很好的交互式接口。例如，用户可以要求列出远程机器上的特定目录下的所有文件。另外用户还可以通过 Help 命令来获取可以使用的 FTP 命令的帮助信息。

2．格式（表示）指定

FTP 允许客户端指定数据存储的类型和格式。例如，用户可以指定一个文件是否包含文本或二进制整数以及文本文件采用的是 ASII 码集还是 EBCDIC 码集。

3．认证控制

FTP 要求客户端通过发送用户名和口令来获得文件访问的授权。如果不能提供有效的用户名和口令将被拒绝访问服务器。

13.1.6 FTP 模型

和其他服务器一样，大多数的 FTP 服务器实现都允许同时被多个客户端所访问。客户端通过 TCP 协议（Telnet）连接到服务器。服务器端的 FTP 解释器进程接受并管理客户端发起的控制连接，同时启动另一个数据传输连接进程。控制连接接受用户的命令并告知服务器传输哪些文件，数据传输连接仍使用 TCP 作为传输协议来传输所有的数据。在客户端和服务器端使用 FTP 的模型如图 13.1 所示。

在图 13.1 所示的模型中，涉及的相关的名词解释如下：

控制连接：服务器和客户端的 FTP 协议解释器间传递命令及回复的通信链路，该连接基于 Telnet 协议。

数据连接：负责实际数据传输的全双工通信链路，有其特定的模式和类型，其传输的数据可以是一个文件的一部分，也可以是整个文件或多个文件，该连接可以存在于一个服务器数据传输进程和一个客户端数据传输进程之间，也可以存在于两个服务器数据传输进程之间。

数据端口：被动的数据传输进程通过监听该端口来获取主动数据传输进程的连接请求以便打开数据连接。

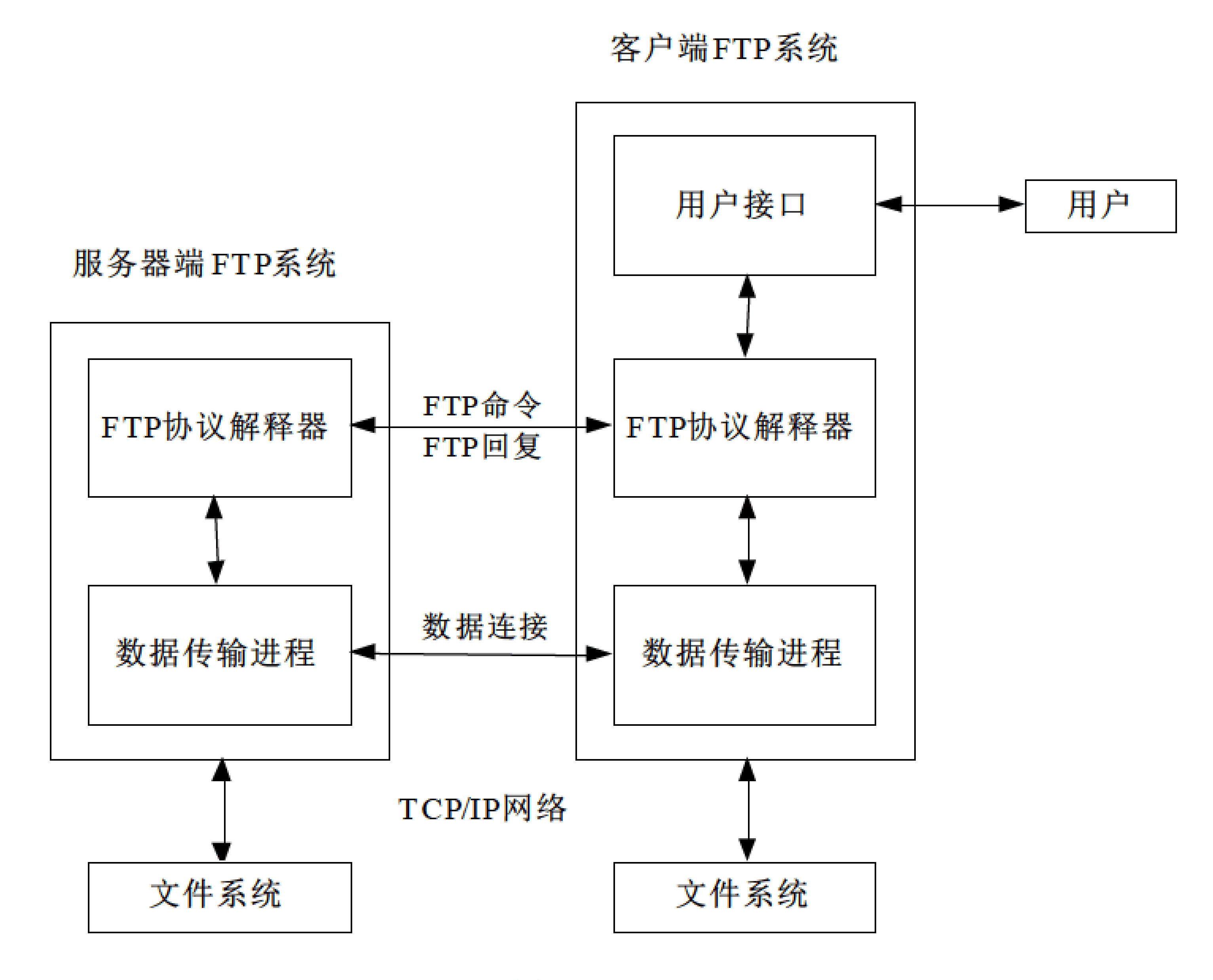
数据传输进程：数据传输进程用以建立和管理数据连接，可以是主动模式的也可是被动

模式的。

协议解释器：协议在客户端和服务器端的功能是完全不一样的，所以客户端和服务器端的协议解释器也很不一样。

服务器端数据传输进程：在通常的主动模式下，它通过监听数据端口来建立数据连接，为数据传输和存储设置参数并根据其协议解释器的命令进行数据传输。它也可设置为被动模式来获取数据连接而不是初始化一个数据连接。

用户：希望获得文件传输服务的人或者进程。



注：1. 数据连接可以是任一方向的

2. 数据连接并不存在于 FTP 交互的全过程

图 13.1　FTP 使用模型

从图中可以看出，客户端向服务器发起的控制连接为一个 TCP 连接，而与之相关的数据传输连接为另一个 TCP 连接。通常控制连接要一直持续到两者之间的 FTP 会话结束，而数据传输连接是每次文件传输前建立，传送完毕后即撤销。总的来说，数据传输连接以及使用该连接的数据传输进程是按照需要动态建立的，而控制连接要贯穿整个 FTP 会话的始末。一旦控制连接撤销，则 FTP 会话中断，同时两边的所有数据连接中断。

13.1.7　TCP 端口号的分配

当一个客户端向服务器发起一个连接时，客户端本机使用一个随机分配的协议端口号，而通过众所周知的服务器 21 端口来与之通信。我们知道，一个服务器可以使用同一个端口来与多个不同的客户端通信，因为 TCP 可以根据终端的不同来区分各个连接。而当需要新建一个用于数据传输的 TCP 连接时，显然它们将不可以使用控制连接中的原有端口，这时客户

端重新获取一个未被使用的 TCP 端口，而服务器端则使用系统预留的 20 端口来进行 FTP 的数据传输。为了确保服务器端的数据传输进程确实是和客户端的正确的数据传输进程交互，服务器端必须拒绝从未知的进程发起的数据连接。这样，当它响应 TCP 连接请求时，服务器需要自己指定通信时使用的客户端端口号。这时我们就能够理解为什么协议需要使用两个 TCP 连接了，首先客户端协议解释器申请一个本地端口用于文件传送，并在本地生成一个传送进程侦听该端口，然后通过控制连接把该端口号传给服务器，最后等待服务器向该端口建立一个 TCP 连接。也就是说，除了传送 FTP 命令外，控制连接还被用来统一协调客户端和服务器之间动态数据传输连接建立及端口号指定。

FTP 在控制连接上交互的数据使用的格式并没有采用新的规范，而是采用了网络虚拟终端协议 Telnet。然而和完整的 Telnet 协议不同的是，FTP 不允许在命令中包含可选参数，而是只使用了基本的网络虚拟终端定义。这样，对 FTP 控制连接的管理要比标准的 Telnet 连接简单得多。通过直接使用 Telnet 协议简化了 FTP 协议所要考虑的问题域。

13.1.8　基本的客户端-服务器交互

客户端和服务器端的交互一般是这样，客户端向服务器发送命令，服务器须向客户端发送回复，命令和回复通过控制连接完成，命令是大小写无关的，而回复包括如下内容：3 位长的数字代码和文本，数字用于向程序指示状态而文本向用户指示状态。回复也可以是多行的。可以看到整个交互过程中控制连接本身不传送任何数据。

对文件传输协议命令的回复主要是为了实现文件传输过程中请求与相应的同步，确保客户端进程知道服务器的当前状态。每个客户端传出的命令至少应能得到一个回复，有些得到的是多个回复，这时多个回复必须很容易地被区分。另外，一些命令是有序的，例如 USER，PASS 和 ACCT，或者 RNFR 和 RNTO。对它们的回复反映了如果所有的命令得以成功执行的中间状态。该系列命令中任一个的执行失败将需要从头把它们重新执行一遍。每个 FTP 回复包括 3 位长的数字码并紧跟着一段文字说明。数字被设计用来自动确定下一步所要进入的状态，文本主要是用于人为操作。实际上，数字已经包括了足够的编码信息，所以用户端协议解释器不需要检查其后所带的文本并且可以选择是否发送给用户。另外，这些文本也是和各个 FTP 服务器相关的，对同一个回复码文本可能是不一样的。

一个回复被定义为包括 3 位长的数字码后跟一个空格，然后是一行文本（通常最大行长度已规定好），最后是 Telnet 的行终结符。不过有时文本的长度不止一行，这时整个文本需要被标记起来，这样客户端进程就知道什么时候可以暂停读取回复（例如：暂时停止处理控制连接上的输入以处理其他事情）。这就需要回复的第一行做些格式处理来表明将有更多的文本行，而在最后一行也需要特定的格式来指明这是最后一行，这其中至少要有一行包括适当的回复码来指示当前处理的状态。为了尽可能减少冲突，第一行和最后一行的回复码必须一致。这样对于多行的回复其格式应该是这样，第一行以原本的回复码开始，后面立即跟一个连字符“-”然后跟文本，最后一行以同样的回复码开始，后面跟一个空格，然后可以有一些文本，最后以 Telnet 行结束符来结束。

例：

123-第一行



第二行
234 以数字开始的另一行
123 最后一行

13.1.9 FTP 命令

13.1.9.1 访问控制命令

以下命令用于指定控制标识（命令码在括号中表示）。

❒ 用户名（USER）

参数域是一个 Telnet 字符串用以标识一个用户，用户标志是服务器提供文件访问所必需的。这个命令通常是控制连接建立后用户需传送的第一个命令，一些服务器可能还要求通过口令和账号命令的形式来提供一些更多的信息。 服务器也可以允许在任何时候使用一个新的用户名命令来改变控制和账号信息。这样可以清空所有已经提供的用户名、口令和账号信息并重新开始登录过程。所有的传输参数都保持不变并且正在传输的所有的文件可以在原有控制参数下传输结束。

❒ 口令（PASS）

参数域是一个 Telnet 字符串用以给出用户的口令。对一些站点来讲，该命令必须在用户名命令结束后立即执行，以完成用户的鉴别获取访问控制权。因为口令信息是比较敏感的，有必要采取一些措施隐藏或者不打印出来，而 FTP 服务器没有一个好的简单可靠的方法来实现这一点，所以隐藏敏感的密码信息就成为客户端 FTP 进程的一个任务。

❒ 账号（ACCT）

参数域是一个 Telnet 字符串用以给出用户的账号。这个命令和用户名命令没有必然的关联性。一些站点可能需要用户提供账号信息来登录而另一些则只有请求一些特殊访问时才需要，如上传文件。对于后者该命令可以在任何时候执行。

为了区分开不同的状况以便自动化处理，系统提供了这些数字码：当账号被要求用来登录时对口令命令成功的提示码是 332，如果登录不需要账号信息，口令命令成功的回复码是 230。而如果账号信息是将被以后会话中的某个命令所需用到的，服务器会依据是否保存该命令信息而返回 332 或者 532。

❒ 改变工作目录（CWD）

❒ 转至上层目录（CDUP）

转至上层目录是改变工作目录命令的特例，该命令用以简化在对上层目录用不同符号表示的操作系统间进行文件传输的实现。

❒ 结构挂载（SMNT）

该命令允许用户挂载一个不同的文件系统的数据结构而不需要改变他的登录信息和账号信息。传输参数基本不变，关键在于用于指定一个目录或系统的名称跟具体的文件系统有关。

❒ 重新初始化（REIN）

该命令将挂起一个用户，清空所有的输入输出和账号信息，不过允许正在传输的进程正常完成。所有的参数都将复位到初始状态而控制连接仍然处于打开状态。这对于用户及时发

现自己所处的状态是很重要的。在此之后一般紧跟着的是用户名命令。

❐ 注销（QUIT）

该命令执行时如果传输不正在进行将注销一个用户，并关闭控制连接。如果文件传输正在进行，连接将一直等到最后的结果返回然后再被关闭。如果用户进程正在为多个用户传输数据，但不希望一下子全部关闭然后再逐一打开，这时可以使用重新初始化命令来替代注销命令。另外，当控制连接遇到突然的关闭时也将导致服务器主动采取中止（ABOR）命令和注销（QUIT）命令。

13.1.9.2　传输参数命令

所有的传输参数都有默认值，传输参数命令只有在需要改变参数的默认值时才需要被使用。默认值是最后被给定的值，或者说如果参数没有值被指定，将用标准的默认值来替代。这就意味着服务器必须"记得"应用的默认值。这些命令可以以任何顺序给出，不过必须在 FTP 服务请求之前。以下命令被用来指定传输参数。

❐ 数据端口（PORT）

这个参数用来指定数据连接中的主机端口号。对于客户端和服务器来讲数据端口都是有默认值的，所以在通常情况下并不需要使用该命令及其回复。该命令被使用时，其参数域的值应该是一个 32 位的因特网主机 IP 地址加上一个 16 位的主机端口号的字串，地址信息分为 8 位一组以十进制表示，组间以逗号分隔。一个数据端口命令的例子如下：

PORT h1,h2,h3,h4,p1,p2

这里 h1 是因特网地址的高 8 位。

❐ 被动（PASV）

该命令要求服务器端的数据传输进程侦听数据端口（默认数据端口），当收到一个传输命令之后它将在该端口等候一个连接而不是主动初始化一个连接。对该命令的响应包括服务器正在侦听的主机和端口地址。

❐ 表示类型（TYPE）

后面跟的参数用来指定数据表示和存储的格式类型。有些类型可能还有第二个参数。第一个参数用一个 Telnet 字符表示，用以指示文本是 ASCII 或是 EBCDIC 编码，第二个参数为一个十进制整数用以指示字节的大小。两个参数之间用一个空格键分隔。

❐ 数据结构（STRU）

其参数为一个 Telnet 字符用以指示数据存储和表示的格式。

以下代码用来指定数据格式：

F – 文件（无记录结构）R – 记录结构 P – 页面结构

默认的结构类型为文件。

❐ 传输模式（MODE）

其参数是一个 Telnet 字符用来指定传输的模式。

以下代码用以指定文件传输模式：

S – 数据流式 B – 数据块式 C – 压缩模式

默认的数据传输模式是数据流式。

13.1.9.3 FTP 服务命令

FTP 服务命令用来指定用户请求的文件传输及文件系统的相关功能。FTP 服务命令的参数通常是一个文件路径。文件路径的语法必须符合服务器端的格式规范（具有标准的默认普适性）和控制连接的语言规范。默认的处理方式是使用最终被确定的盘符、目录或文件名或是本地用户的标准默认设置。这些命令可以以任何次序执行，不过获得重命名命令后面需要跟重命名命令，而重新启动命令后须跟中断服务命令（如：STOP 或 RETR）。文件数据将根据 FTP 服务命令的要求通过数据传输连接发送，而一些特定的回复信息则通过控制连接发送。以下命令用来指定 FTP 服务请求。

❒ 获取（RETR）

该命令将要求服务器端的数据传送进程发送参数中所指定路径的文件的一份复制至数据传输连接另一端的客户端。而服务器端的该文件的内容和状态都不会发生改变。

❒ 存储（STOR）

该命令将要求服务器端的数据传输进程接受数据连接传送过来的数据，并按指定的路径把它们存储至服务器端。如果路径中指定的位置已经存在同名文件则会被传输来的数据所替代，而如果该位置原来没有同名文件则会创建一个新的文件。

❒ 惟一存储（STOU）

❒ 附加（连同创建）（APPE）

该命令将使服务器端的数据传输进程从数据连接中接受数据然后在服务器端存储该数据，如果指定的路径上已经有同名的文件则数据被附加到原有文件上，否则就新建一个数据文件。

❒ 分派（ALLO）

❒ 重新启动（REST）

参数域指示服务器端哪一个文件传输进程需要被重启。该命令不会引起文件传输，而是会跳转至文件的某个特定断点，该命令后面须跟随适当的 FTP 服务命令来使传输恢复。

❒ 中止（ABOR）

该命令让服务器取消前一个 FTP 服务命令的执行以及与其相关的数据传输。

❒ 删除（DELE）

❒ 删除目录（RMD）

❒ 新建目录（MKD）

❒ 输出工作目录（PWD）

❒ 文件清单（LIST）

该命令使得服务器发送一份文件列表到被动的数据传输进程。如果在参数域的路径指定了文件目录或某个文件组，服务器将把该目录下的所有文件名列出来，如果路径是指向一个确定的文件的，服务器将给出该文件的具体信息。如果参数域中不给定路径，那么默认路径为当前工作目录。这些数据以 ASCII 码或 EBCDIC 码的形式在数据连接上传输（用户必须自己确认究竟是 ASCII 码还是 EBCDIC 码）。文件的具体信息因系统的不一样而会有很大的变化，所以这些信息对程序自动化处理没有多大意义，不过对人来说却很有价值。

❒ 名称列表（NLST）

该命令将使服务器把目录列表发给客户端的站点，参数域的路径指定了文件目录或某个

文件组，如果参数域中不给定路径，那么默认路径为当前工作目录。服务器将返回一串文件名而不包含其他任何信息。这些数据以 ASCII 码或 EBCDIC 码的形式在数据连接上传输（用户仍需确保编码格式正确）。该命令被用来返回一些信息以便程序可以自动作进一步的文件处理，比如多个文件获取操作等。

❒ 站点参数（SITE）

该命令被服务器端用来向外提供根本系统密切相关的一些服务说明信息，这些信息对文件传输来讲是至关重要的，但还不是完备的。可以通过 HELP SITE 命令获取更多有关服务器文件服务参数和语法的规格说明。

❒ 系统（SYST）

该命令被用来检查服务器端的操作系统信息，其回复将包括操作系统名称的第一单词。

❒ 状态（STAT）

该命令的执行结果为客户端将得到一个从控制连接发回的有关当前状态的回复。该命令可以在进行文件传输时执行以返回传输的进展状况，也可以在文件传输的间隙执行该命令。对于后者来说可以附带参数，如果参数为一个路径，那么其执行的结果类似于 LIST 命令，不同的是数据是从控制连接发送的。如果只给出部分路径，那么服务器将返回与之相关的文件名列表或是属性信息。如果不给任何参数，服务器将返回通用的 FTP 服务状态信息，其中包括所有的传输参数和连接状态。

❒ 帮助（HELP）

该命令将使得服务器根据自身的实现方式通过控制连接发送一些有用的信息给用户。它也可以带参数（如任何的命令名），将可以得到从控制连接返回的更为详尽的信息。回复码为 211 或者 214。RFC 文档建议 HELP 命令可以在 USER 命令执行前使用。程序可以通过这些反馈自动设置一些与站点相关的参数，如执行 HELP SITE。

❒ 打探（NOOP）

该命令不会对任何参数或前继命令产生影响，也不产生任何的实际动作，只是等待服务器发送一个 OK 回复。

13.1.10　FTP 用户会话样例

在用户看来 FTP 是一个交互式的系统，一旦被调用，客户端将不断重复如下操作：读取一行输入，解析该行以获取命令字及其参数，使用给定的参数执行该命令。下面给出一个在 Windows 命令行窗口下使用 FTP 的样例。

```
xiaopeng-> ftp 211.65.59.99
Connected to 211.65.59.99.
220 Serv-U FTP Server v4.0 for WinSock ready...
user xiaopeng
331 User name okay, need password.
PASS ********
230 User logged in, proceed.
LIST
150 Opening ASCII mode data connection for /.
226 Transfer complete.
```

```
pwd
257 "/" is current directory.
retr /My lessons/多播.PDF
200 PORT command okay.
150 Opening BINARY mode data connection for /My lessons/多播.PDF (1176145 bytes).
226 Transfer complete.
Quit
221 Goodbye
```

从上面的交互中，可以看出以上介绍的部分命令的用法及相应的服务器系统回复状况。值得注意的是在输出中有一个 PORT 命令，客户端的 PORT 命令用来报告有一个新的端口已被获取用来进行数据传输。客户端把端口号通过控制进程传输至服务器端，但建立一个新的数据连接时两端便都使用该端口号。数据传输结束后两端的数据传输进程关闭数据连接。

13.2 TFTP

尽管 FTP 是在 TCP/IP 协议族中最通用的文件传输协议，但它同时也是最复杂的并且给编程带来了不便。许多应用往往并不需要 FTP 协议所提供的全部功能，也支持不了其复杂性。例如 FTP 要求客户端和服务器必须要具备同时管理多个 TCP 连接的能力，而这对一些不具备复杂操作系统的个人计算设备来讲是很难实现的。

TCP/IP 协议族中还包括另一个文件传输协议，它可提供简单的花费很小的文件传输服务，这就是 TFTP（Trivial File Transfer Protocol），它主要是针对那些在客户端和服务器将不需要复杂交互的应用而设计的。TFTP 把操作限制在简单的文件传输而不提供用户认证和授权等功能。正是因为这些功能上的局限，TFTP 的软件要比 FTP 小很多。

对许多应用来讲，软件的大小显得很重要，比如无驱设备的制造商就可以把 TFTP 的软件烧制在板载只读存储器（ROM）中并在机器加电时加载到内存中。存放在 ROM 的程序称作引导程序，使用 TFTP 的好处就在于它允许引导程序使用与操作系统一样的底层 TCP/IP 协议，这样一台计算机就可以在另一个物理位置上利用文件服务器上的引导程序启动起来。

和 FTP 协议不同，TFTP 协议不需要一个可靠的流式数据传输服务。它工作在 UDP 或其他不可靠的包传递服务之上，通过超时重发来确保文件到达对方。发送方以固定大小（512 字节）的数据块发送数据，对每个数据块都是等到收到到达回复之后再发送下一个数据块。相应地接受方每收到一个数据块都发送回复。

TFTP 的规则是比较简单的。发送的第一个数据包发送一个文件传输请求，并在客户端和服务器之间建立起交互，该数据包指明了传送文件的名称以及文件是被读取（传送到客户端）还是被写入（传送到服务器端）。被传送的文件数据块从 1 开始顺序编号，每个数据包头中都包含它所传数据所属的数据块号，而接受方发送的回复中也包含相应的数据块号。最后一个不满 512 字节的数据包表示着文件传输的结束。另外，在数据和回复中都可以发送出错信息，一旦检测到出错信息，传输将被终止。

一旦一个读取或者写入的请求被发送，服务器端将使用 IP 地址和 UDP 端口号来标志一个后续的操作。这样后面的传送数据块的数据包和回复的数据包都不需要在包中包含文件的

名称信息。如果得到一个块丢失的信息，将使得该数据块被重新发送一次，而在其他出错情况下则简单地引发文件传输会话的终止。

TFTP 的特别之处在于它是对称的，收发双方使用的都是超时重发机制。如果发送方发送数据超时，它将把最后一个数据块重新发送一次，而如果接受方发送回复超时，它也会将回复重新发送一次。这种双方同时检测超时的机制确保了传输不会因为单个数据包的丢失而宣告失败。

尽管这种对称的方式保证了文件传输的健壮性，但是它也会有负面的影响，可能会导致过多的数据包重发。已经发现的问题是在最坏的情况下可能会进入死循环并且每个数据包都会被发送两次。尽管 TFTP 存在这个缺陷，但是能够满足最小的文件传输需求，并且可以支持多文件类型。另外它还可以和 E-mail 服务集成在一起，可以把文件当作邮件来发送。

13.3　NFS

NFS 最初由 SUN 微系统公司开发，全称为网络文件系统（Network File System），提供在线透明的集成的文件共享服务。许多站点都采用它来构建文件系统。从用户的观点来看，NFS 是感觉不到的，用户可以使用任意地方的程序并使用任何文件作为输入和输出。从文件名称本身不能看出一个文件是本地的还是远程的。

当一个应用程序执行时，它调用操作系统来打开一个文件，或者读取存储文件中的数据。文件访问机制接受该请求并根据文件是在本地磁盘还是远程机器把任务自动交付给本地文件系统或 NFS 客户端。当它接受到一个请求时，客户端软件使用 NFS 协议跟远程主机的对应的服务器通信并执行所请求的操作。当远程服务器响应后，客户端把结果返回给应用程序。

设计者在构造 NFS 协议时把它分成三个相对独立的部分：NFS 协议本身、一个通用的远程过程调用（RPC）机制和一个通用的扩展数据表示机制（XDR）。他们的初衷是把三者分开以便其他应用和协议可以独立地使用 RPC 和 XDR。

从程序员的观点来看，NFS 本身不提供任何新的程序可调用的方法。一旦一个管理员配置好一个 NFS 系统，程序访问远程文件的操作和本地文件是完全一样的。但是，RPC 和 XDR 提供了可供程序员使用的用以构建分布式程序的机制。例如，程序员可以把程序分为客户端和服务器端两部分，两者之间以 RPC 作为主要的通信机制。在客户端，程序员设计一些过程作为远程的，让编译器植入 RPC 代码到这些过程中。在服务器端，程序员实现相应的过程并使用另一些 RPC 机制声明它们是服务器的一部分。当客户端程序调用远程的某个过程时，RPC 自动收集数据值和变量形成一个消息，把该消息发送至远程服务器并等待回复，然后把返回的值存入到恰当的参数中。在此过程中，与远程服务器的通信基本上随远程方法调用而自动进行的。RPC 机制隐藏了协议的细节，使得那些即使对底层通信协议知之甚少的程序员也可以写出分布式程序来。

另一个相关的工具是 XDR，它使得程序员在异构系统之间传输数据而不需要写任何转换硬件数据表示的过程。例如，并不是所有的机器表示 32 位二进制数的格式都是一样的。有些把数据的高位字节存放在存储器的高地址处，而另一些则把它们放在低地址处，这样如果程序员使用网络把代表一个整数的字节从一台机器移至另一台机器而不重新排序则数据的



值就会发生变化。XDR 通过定义与机器无关的表示格式而解决了这个问题。在数据传输的一端，程序通过调用 XDR 过程把本地的硬件数据表示转化为机器无关的数据表示。一旦该数据需要被传送到另一台机器，则接收端的程序通过调用 XDR 过程把机器无关的数据表示再转换为本地的数据表示格式。

XDR 的最大优势在于它自动完成了绝大多数的数据格式转换工作。程序员并不需要手工调用 XDR 过程。实际上他们只需要在程序中向 XDR 编译器指明哪些数据需要被转化，那么编译器将自动生成包含 XDR 库过程调用的程序。

第 14 章 SNMP 网络管理体系结构

随着网络技术的飞速发展，网络的数量也越来越多。网络中存在着大量的诸如工作站、服务器、网卡、路由器、网桥和集线器等设备。如何管理这些设备就变得十分重要。

要对网络做出适当的管理，网络管理员首先需要了解网络中各种设备及链路的工作状态，这需要网络管理员维持一个网络管理信息库（MIB）以及时了解网络中各种设备的工作状态并做出相应的决策。由于互联网规模很大，尤其是全球因特网，其触角已延伸到全世界大多数国家的许多地方，而各种被管设备又分散在网络的各个角落，由网络管理员手工维护管理信息库几乎是不可能的，必须借助管理软件来获取所有被管设备的状态信息。这就要求管理软件有与被管设备进行通信的能力，因此需要有相应的网管通信协议。另一方面，网络管理信息库的获取及维护需要被管设备的支持，即被管设备必须能以一定的方式提供有关状态信息。目前，设备制造商一般都在网卡、路由器、网桥等网络设备中提供网管功能，这些设备一般都能主动或被动地提供相关信息。但由于不同的厂商提供的信息格式和存储方式千差万别，要使得网络管理员准确获取并理解各种状态信息还需有一套关于管理信息结构的统一约定。

因此，一个网管系统至少具备下述要素：管理员和代理；管理信息库 MIB（Management Information Base），管理信息库包含所有可被查询和修改的参数。RFC 1213[McCloghrie and Rose 1991]定义了第二版的 MIB，叫做 MIB-II；管理信息结构 SMI（Structure of Management Information）是关于 MIB 的一套公用的结构和表示符号，这在 RFC 1155 [Rose and McCloghrie 1990]中进行了定义；通信协议，目前 Internet 上主要使用简单网络管理协议 SNMP（Simple Network Management Protocol）。RFC 1157 [Case et al. 1990]定义了简单网管协议。本章主要介绍网管系统中的一些基本概念，网管系统的工作模式。

14.1 SNMP 体系结构

14.1.1 TCP/IP 网络管理的发展

在 TCP/IP 的早期开发中，网络管理问题并未得到太大的重视。直到 20 世纪 70 年代，还一直没有网络管理协议，只有互联网络控制信息协议（ICMP）可以作为网络管理的工具。ICMP 提供了从路由器或其他主机向主机传送控制信息的方法，可用于所有支持 IP 的设备。从网络管理的观点来看，ICMP 最有用的特性是回声（echo）和回声应答（echo reply）消息对。这个消息对为测试实体间能否通信提供了一个机制。echo 消息要求其接收者在 echo reply 消息中返回接收到的内容。另一个有用的消息对是时间戳（timestamp）和时间戳应答（timestamp reply），这个消息对为测试网络延迟特性提供了机制。

与各种 IP 头选项结合，这些 ICMP 消息可用来开发一些简单有效的管理工具。典型的例

子是广泛应用的分组互联网络探索（PING）程序。利用 ICMP 加上另外的选项如请求间隔和一个请求的发送次数，PING 能够完成多种功能。包括确定一个物理网络设备能否寻址，验证一个网络能够寻址和验证一个主机上的服务器操作。

PING 在一些工具的配合下满足了 TCP/IP 网络初期的管理要求。但是到了 20 世纪 80 年代后期，当互联网络的发展呈指数增加时，人们感到需要开发比 PING 功能更强并易于普通网络管理人员学习和使用的标准协议。因为当网络中的主机数量上百万，独立网络数量上千的时候，已不能只依靠少数网络专家解决管理问题了。

1987 年 11 月发布了简单网关监控协议（SGMP），成为提供专用网络管理工具的起点。SGMP 提供了一个直接监控网关的方法。随着对通用网络管理工具需求的增长，出现了 3 个有影响的方法。

（1）高层实体管理系统（HEMS）：主机监控协议（HMP）的一般化。

（2）简单网络管理协议（SNMP）：SGMP 的升级版。

（3）TCP/IP 上的 CMIP（CMOT）：最大限度地与 OSI 标准的 CMIP、服务以及数据库结构保持一致。

1988 年，互联网络活动会议（IAB）确定了将 SNMP 作为近期解决方案进一步开发，而把 CMOT 作为远期解决方案的策略。当时普遍认为：TCP/IP 不久将会过渡到 OSI，因而不应在 TCP/IP 的应用层协议和服务上花费太多的精力。SNMP 开发速度快，并能为网络管理经验库的开发提供一些基本的工具，可用来满足眼前的需要。

为了强化这一策略，IAB 要求 SNMP 和 CMOT 使用相同的被管对象数据库。即在任何主机、路由器、网桥以及其他管理设备中，两个协议都以相同的格式使用相同的监控变量。因此，两个协议有一个公共的管理信息结构（SMI）和一个管理信息库（MIB）。

但是，人们很快发现这两个协议在对象级的兼容是不现实的。在 OSI 的网络管理中，被管对象是很成熟的，它具有属性、相关的过程、通报以及其他一些与面向对象有关的复杂的特性。而 SNMP 为了保持简单性，没有这样复杂的概念。实际上，SNMP 的对象在面向对象的概念下根本就不能称为对象，它们只是带有一些如数据类型、读写特性等基本特性的变量。因此 IAB 最终放松了公共 SMI/MIB 的条件，并允许 SNMP 独立于 CMOT 发展。

从对 OSI 的兼容性的束缚中解脱后，SNMP 取得了迅速的发展，很快被众多的厂商设备所支持，并在互联网络中活跃起来。而且，普通用户也选择了 SNMP 作为标准的管理协议。

SNMP 最重要的进展是远程监控（RMON）能力的开发。RMON 为网络管理者提供了监控整个子网而不是各个单独设备的能力。除了 RMON，还对基本 SNMP MIB 进行了扩充。有些扩充采用标准的网络接口，例如令牌环（token ring）和光纤分布数据接口（FDDI），这种扩充是独立于厂商的。

但是，单靠定义新的或更细致的 MIB 扩充 SNMP 是有限的。当 SNMP 被用于大型或复杂网络时，它在安全和功能方面的不足就变得明显了。为了弥补这些不足，1992 年 7 月发表了 3 个增强 SNMP 安全性的文件作为建议标准。增强版与原来的 SNMP 是不兼容的，它需要改变外部消息句柄及一些消息处理过程。但实际定义协议操作并包含 SNMP 消息的协议数据单元（PDU）保持不变，并且没有增加新的 PDU。目的是尽量实现向 SNMP 的安全版本的平滑过渡。

但是这个增强版受到了另一个方案的冲击。同样是在 1992 年 7 月，四名 SNMP 的关键

人物提出一个称为 SMP 的 SNMP 新版本。并实现了四个可互操作的方案。两个是商业产品，两个是公开软件。SMP 在功能和安全性两方面提高了 SNMP，特别是 SMP 增加了一些 PDU。所有的消息头和安全功能都与提议的安全性增强标准相似。最终 SMP 被接受为定义第二代 SNMP 即 SNMPv2 的基础。1993 年安全版 SNMPv2 发布。

经过几年试用以后，IETF（Internet Engineering Task Force）决定对 SNMPv2 进行修订。1996 年发布了一组新的 RFC（Request For Comments），在这组新的文档中，SNMPv2 的安全特性被取消了，消息格式也重新采用 SNMPv1 的基于“共同体（community）”概念的格式。

删除 SNMPv2 中的安全特性是 SNMPv2 发展过程中最大的失败。主要原因是厂商和用户对 1993 版的 SNMPv2 的安全机制不感兴趣，同时 IETF 要求的修订时间也非常紧迫，设计者们来不及对安全机制进行改善，甚至来不及对存在的严重缺陷进行修改。因此不得不在 1996 年版的 SNMPv2 中放弃了安全特性。

1999 年 4 月 IETF SNMPv3 工作组提出了 RFC2571～RFC2576，形成了 SNMPv3 的建议。目前，这些建议正在进行标准化。SNMPv3 提出了 SNMP 管理框架的一个统一的体系结构。在这个体系结构中，采用 User-based 安全模型和 View-based 访问控制模型提供 SNMP 网络管理的安全性。安全机制是 SNMPv3 的最具特色的内容。

14.1.2　SNMP 基本框架

1．网络管理体系结构

SNMP 的网络管理模型包括以下关键元素：管理站、代理者、管理信息库、网络管理协议。管理站一般是一个分立的设备，也可以利用共享系统实现。管理站被作为网络管理员与网络管理系统的接口。它的基本构成为：

- 一组具有分析数据、发现故障等功能的管理程序。
- 一个用于网络管理员监控网络的接口。
- 将网络管理员的要求转变为对远程网络元素的实际监控的能力。
- 一个从所有被管网络实体的 MIB 中抽取信息的数据库。

网络管理系统中另一个重要元素是代理者。装备了 SNMP 的平台，如主机、网桥、路由器及集线器均可作为代理者工作。代理者对来自管理站的信息请求和动作请求进行应答，并随机地为管理站报告一些重要的意外事件。

与 CMIP 体系相同，网络资源也被抽象为对象进行管理。但 SNMP 中的对象是表示被管资源某一方面的数据变量。对象被标准化为跨系统的类，对象的集合被组织为管理信息库（MIB）。MIB 作为设在代理者处的管理站访问点的集合，管理站通过读取 MIB 中对象的值来进行网络监控。管理站可以在代理者处产生动作，也可以通过修改变量值改变代理者处的配置。

管理站和代理者之间通过网络管理协议通信，SNMP 通信协议主要包括以下能力。

- Get：管理站读取代理者处对象的值。
- Set：管理站设置代理者处对象的值。
- Trap：代理者向管理站通报重要事件。

在标准中，没有特别指出管理站的数量及管理站与代理者的比例。一般地，应至少要有

两个系统能够完成管理站功能，以提供冗余度，防止故障。另一个实际问题是一个管理站能带动多少代理者。只要 SNMP 保持它的简单性，这个数量可以高达几百。

2．网络管理协议体系结构

SNMP 为应用层协议，是 TCP/IP 协议族的一部分。它通过用户数据报协议（UDP）来操作。在分立的管理站中，管理者进程对位于管理站中心的 MIB 的访问进行控制，并提供网络管理员接口。管理者进程通过 SNMP 完成网络管理。SNMP 在 UDP、IP 及有关的特殊网络协议（如 Ethernet、FDDI、X.25）之上实现。

每个代理者也必须实现 SNMP、UDP 和 IP。另外，有一个解释 SNMP 的消息和控制代理者 MIB 的代理者进程。

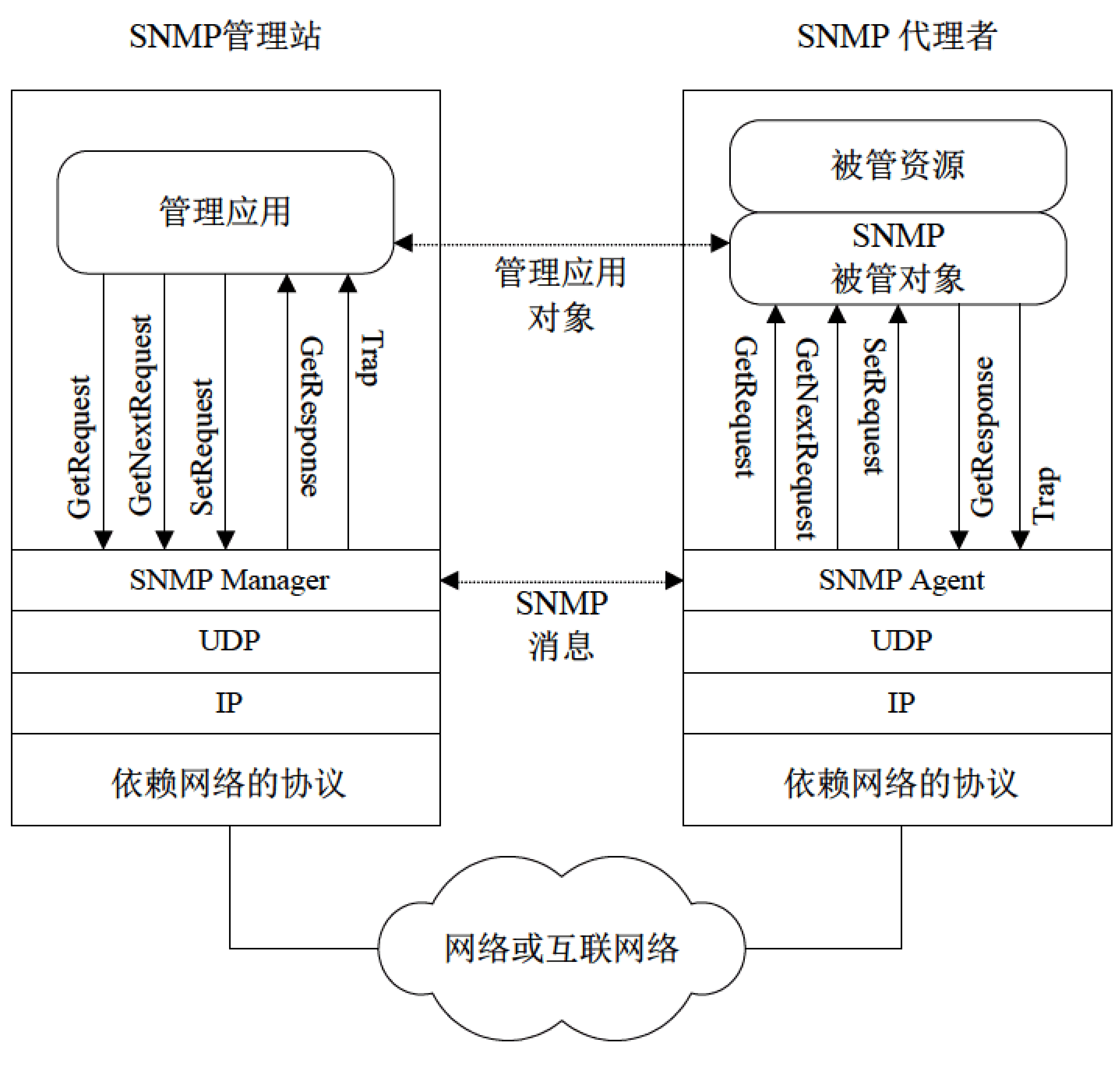


图 14.1 SNMP 的协议环境

图 14.1 描述了 SNMP 的协议环境。从管理站发出三类与管理应用有关的 SNMP 的消息 GetRequest、GetNextRequest、SetRequest。三类消息都由代理者用 GetResponse 消息应答，该消息被上交给管理应用。另外，代理者可以发出 Trap 消息，向管理者报告有关 MIB 及管理资源的事件。

由于 SNMP 依赖 UDP，而 UDP 是无连接型协议，所以 SNMP 也是无连接型协议。在管理站和代理者之间没有在线的连接需要维护。每次交换都是管理站和代理者之间的一个独立的传送。

3．陷阱引导轮询（Trap-directed polling）

如果管理站负责大量的代理者，而每个代理者又维护大量的对象，则靠管理站及时地轮询所有代理者维护的所有可读数据是不现实的。因此管理站采取陷阱引导轮询技术对 MIB

进行控制和管理。

所谓陷阱引导轮询技术是：在初始化时，管理站轮询所有知道关键信息（如接口特性，作为基准的一些性能统计值，如发送和接收的分组的平均数）的代理者。一旦建立了基准，管理站将降低轮询频度。相反地，由每个代理者负责向管理站报告异常事件。例如，代理者崩溃和重启动、连接失败、过载等。这些事件用 SNMP 的 Trap 消息报告。

管理站一旦发现异常情况，可以直接轮询报告事件的代理者或它的相邻代理者，对事件进行诊断或获取关于异常情况的更多的信息。

陷阱引导轮询可以有效地节约网络容量和代理者的处理时间。网络基本上不传送管理站不需要的管理信息，代理者也不会无意义地频繁应答信息请求。

4．代管（Proxies）

利用 SNMP 需要管理站及其所有代理者支持 UDP 和 IP。这限制了在不支持 TCP/IP 协议的设备（如网桥、调制解调器）上的应用。并且，大量的小系统（PC、工作站、可编程控制器）虽然支持 TCP/IP 协议，但不希望承担维护 SNMP、代理者软件和 MIB 的负担。

为了容纳没有装载 SNMP 的设备，SNMP 提出了代管的概念。在这个模式下，一个 SNMP 的代理者作为一个或多个其他设备的代管人。即，SNMP 代理者为托管设备（proxied devices）服务。

图 14.2 显示了常见的一类协议体系结构。管理站向代管代理者发出对某个设备的查询。代管代理者将查询转变为该设备使用的管理协议。当代理者收到对一个查询的应答时，将这个应答转发给管理站。类似地，如果一个来自托管设备的事件通报传到代理者，代理者以陷阱消息的形式将它发给管理站。

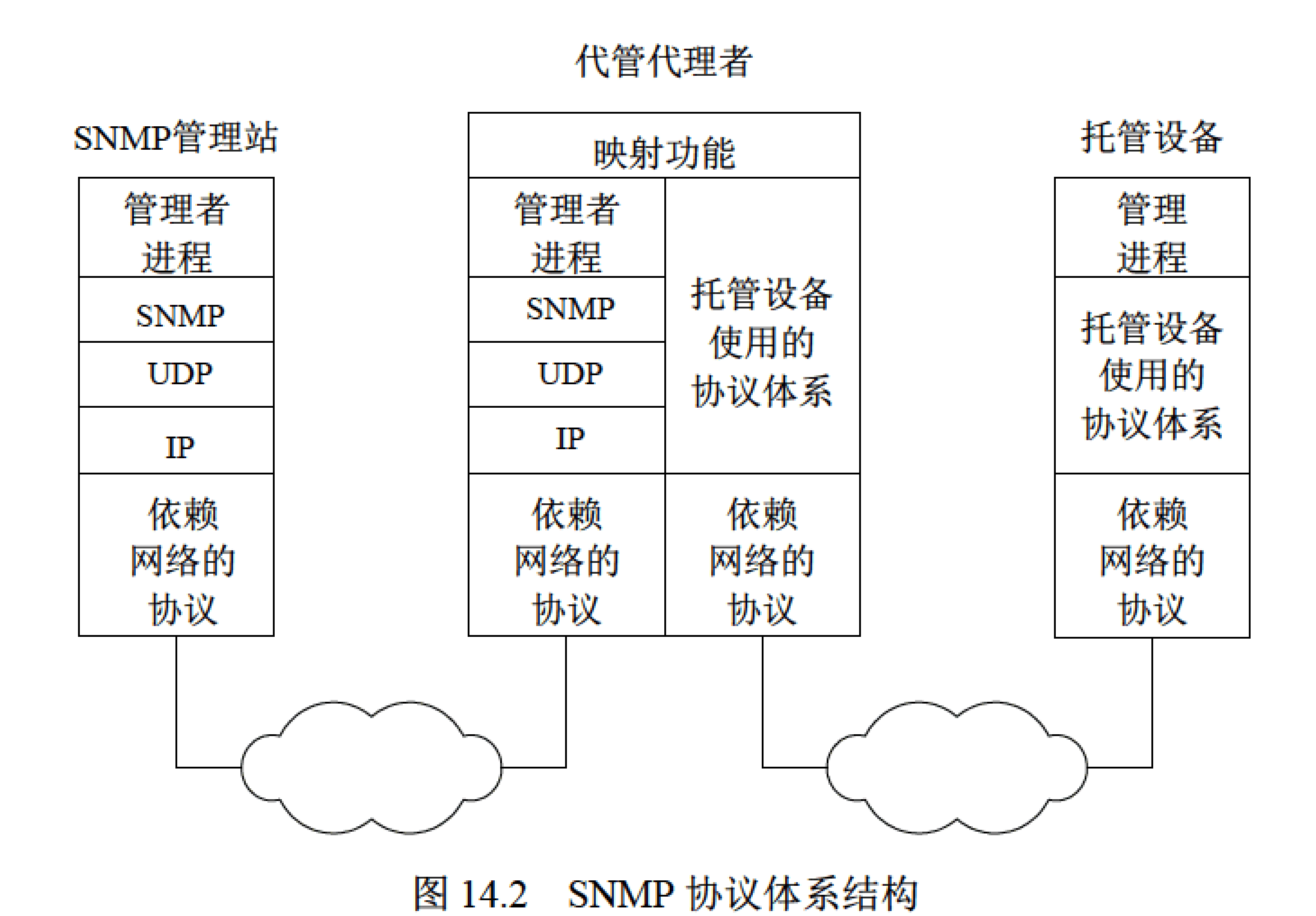


图 14.2　SNMP 协议体系结构

14.2　SNMP 管理信息

与 CMIP 体系相同，SNMP 的基础是包含被管元素信息的被称为 MIB 的数据库。每个被



管资源由对象来表示，MIB 是这些对象的有结构的集合。在 SNMP 中，MIB 本质上是一个树型的数据库结构。网络中每个系统都（工作站、服务器、路由器、网桥等）拥有一个反映系统中被管资源状态的 MIB。网络管理实体可以通过提取 MIB 中的对象值监测系统中的资源，也可以通过修改这些对象值来控制资源。

14.2.1 管理信息结构

SNMP 的规范 SMI（structure of management information）为定义和构造 MIB 提供了一个通用的框架。同时也规定了可以在 MIB 中使用的数据类型，说明了资源在 MIB 中怎样表示和命名。SMI 的基本指导思想是追求 MIB 的简单性和可扩充性。因此，MIB 只能存储简单的数据类型：标量和标量的二维矩阵。我们将看到 SNMP 只能提取标量，包括表中的单独的条目。

SMI 避开复杂的数据类型是为了降低实现的难度和提高互操作性。但在 MIB 中不可避免地包含厂家建立的数据类型，如果对这样的数据类型的定义没有严格的限制，互操作性也会受到影响。

为了提供一个标准的方法来表示管理信息，SMI 必须提供一个标准的技术定义 MIB 的具体结构；提供一个标准的技术定义各个对象，包括句法和对象值；提供一个标准的技术对对象值进行编码。

1．MIB 结构

SNMP 中的所有的被管对象都被排列在一个树型结构之中。处于叶子位置上的对象是实际的被管对象，每个实际的被管对象表示某些被管资源、活动或相关信息。树型结构本身定义一个将对象组织到逻辑上相关的集合之中的方法。

MIB 中的每个对象类型都被赋予一个对象标识符（Object Identifier），以此来命名对象。另外，由于对象标识符的值是层次结构的，因此命名方法本身也能用于确认对象类型的结构。

对象标识符是能够惟一标识某个对象类的符号。它的值由一个整数序列构成。被定义的对象的集合具有树型结构，树根是引用 ASN.1 标准的对象。从对象标识符树的树根开始，每个对象标识符成分的值指定树中的一个弧。从树根开始，第一级有三个节点：iso、ccitt、joint-iso-ccitt。如图 14.3 所示在 iso 节点下面有一个为“其他组织”使用的子树，其中有一个美国国防部的子树（dod）。SNMP 在 dod 之下设置一个子树用于 Internet 的管理。如下 所示：

Internet OBJECT IDENTIFIER :: = { iso (1) org (3) dod (6) 1}

因此，Internet 节点的对象标识符的值是 1.3.6.1。这个值作为 Internet 子树的下级节点标识符的前缀。

SMI 在 Internet 节点之下定义了四个节点：

- ❒ directory 为与 OSI 的 directory 相关的将来的应用保留的节点。
- ❒ mgmt 用于在 IAB 批准的文档中定义的对象。
- ❒ experimental 用于标识在 Internet 实验中应用的对象。
- ❒ private 用于标识单方面定义的对象。

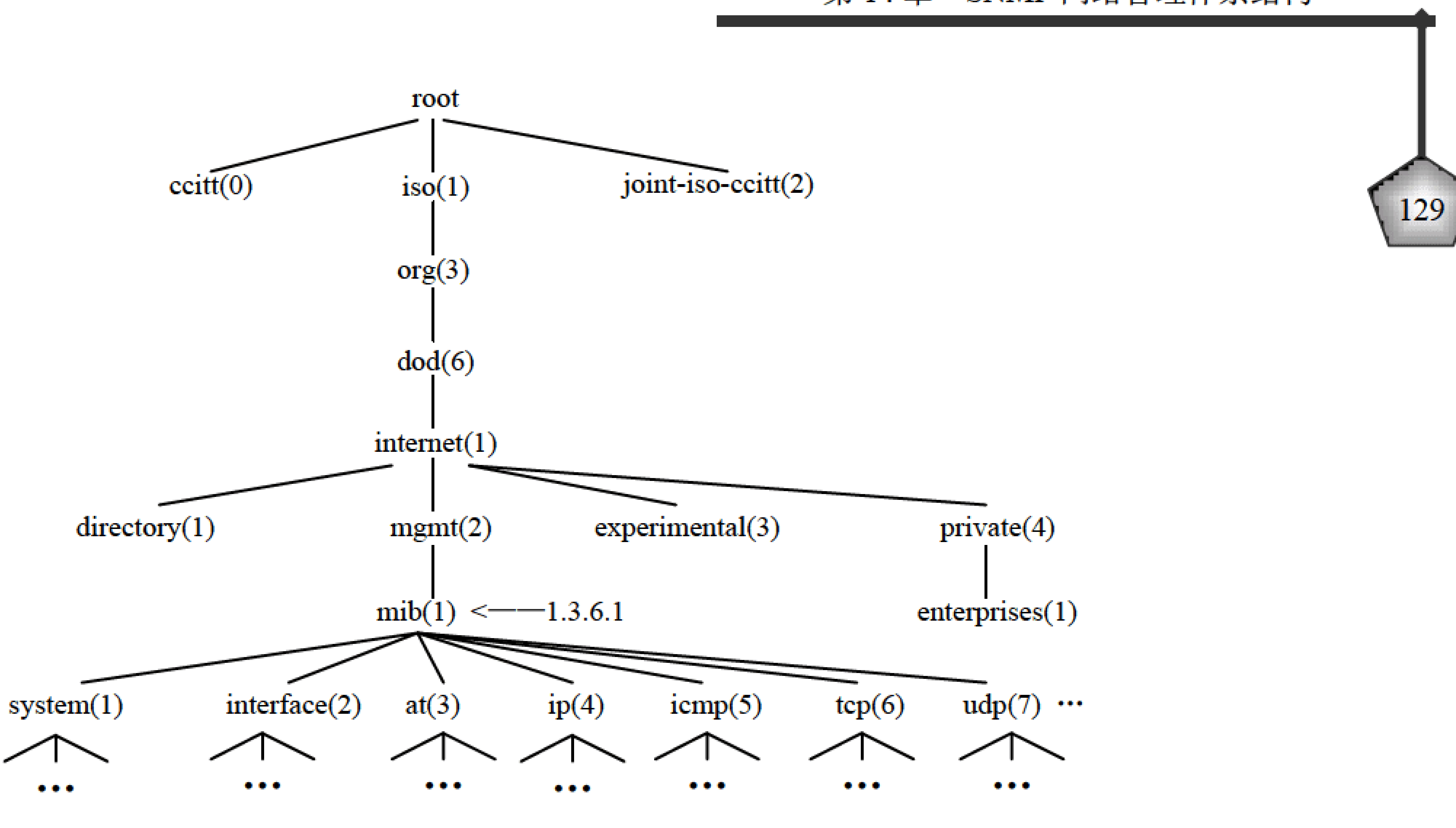


图 14.3　对象标识符树型结构

mgmt 子树包含 IAB 已经批准的管理信息库的定义。现在已经开发了两个版本的 MIB，MIB-1 和它的扩充版 MIB -2。两者子树中的对象标识符是相同的，因为在任何配置中，只有一个 MIB。

MIB 中的 MIB -1 或 MIB -2 以外的对象可以用以下方法定义：

- 由一个全新的修订版（如 MIB -3）来扩充或取代 MIB -2。
- 可以为特定的应用构造一个实验 MIB。这样的对象随后会被移到 mgmt 子树之下。例如定义包含各种传输媒体的 MIB（例如为令牌环局域网定义的 MIB）。
- 专用的扩充可以加在 private 子树之下。

private 子树目前只定义了一个子节点 enterprises，用于厂商加强对自己设备的管理，与用户及其他厂商共享信息。在 enterprises 子树下面，每个注册了 enterprise 对象标识符的厂商有一个分支。

Internet 节点之下分为 4 个子树的做法为 MIB 的进化提供了很好的基础。通过对新对象的实验，厂商能够在其被接受为 mgmt 的标准之前有效地获得大量的实际知识。因此这样的 MIB 既是对管理符合标准的对象直接有效的，对适应技术和产品的变化也是灵活的。这一点也反映了 TCP/IP 协议的如下特性：协议在成为标准之前进行大量的实验性的使用和调测。

2．对象句法

SNMP MIB 中的每个对象都由一个形式化的方法定义，说明对象的数据类型、取值范围以及与 MIB 中的其他对象的关系。各个对象以及 MIB 的整体结构都由 ASN.1 描述法定义。为了保持简单，只利用了 ASN.1 的元素和特征的一个有限的子集。

（1）UNIVERSAL 类型：ASN.1 的 UNIVERSAL 类由独立于应用的通用数据类型组成。其中只有以下数据类型被允许用于定义 MIB 对象：

- integer（UNIVERSAL 2）。
- octetstring（UNIVERSAL 4）。

- null（UNIVERSAL 5）。
- object identifier（UNIVERSAL 6）。
- sequence, sequence-of（UNIVERSAL 16）。

前 3 个是构成其他对象类型的基本类型。

object identifier 惟一标识对象的符号，由一个 integer 序列组成，序列中的 integer 被称为子标识符。对象标识符的 integer 序列从左到右，定义了对象在 MIB 树型结构中的位置。

sequence 和 sequence-of 用于构成表。

（2）APPLICATION-WIDE 类型：ASN.1 的 APPLICATION 类由与特定的应用相关的数据类型组成。每个应用包括 SNMP，负责定义自己的 APPLICATION 数据类型。在 SNMP 中已经定义了以下数据类型：

- networkaddress：该类型用 CHOICE 结构定义，允许从多个协议族的地址格式中进行选择。目前，只定义了 IpAddress 一种地址格式。
- ipaddress：IP 格式的 32 位地址。
- counter：只能做增值不能做减值运算的非负整数。最大值被设为 $2^{32}-1$，当达到最大值时，再次从 0 开始增加。
- gauge：可做增值也可做减值运算的非负整数。最大值被设为 $2^{32}-1$，当达到最大值时被锁定，直至被复位（reset）。
- timeticks：从某一参照时间开始以百分之一秒为单位计算经历的时间的非负整数。当 MIB 中定义的某个对象类用到这个数据类型时，参照时间在该对象类的定义中指出。
- opaque：该数据类型提供一个传递任意数据的能力。数据在传输时被作为 OCTET STRING 编码。被传递的数据本身可以是由 ASN.1 或其他句法定义的任意的格式。

3．定义对象

管理信息库由一个对象的集合构成，每个对象都有一个型和一个值。型是对被管对象种类的定义，因此型的定义是一个句法描述。对象的实例是某类对象的一个具体实现，具有一个确定的值。

怎样定义 MIB 中的对象呢？ASN.1 是将被使用的描述法。ASN.1 中包含一些预定义的通用类型，也规定了通过现有类型定义新类型的语法。定义被管对象的一个可选方法是定义一个被称为 Object 的新类型。这样，MIB 中所有的对象都将是这种类型的。这个方法在技术上是可行的，但会产生定义不便于应用的问题。我们需要多种值的类型，包括 counter、gauge 等。另外，MIB 支持二维表格或矩阵的定义。因此，一个通用的对象类型必须包含参数来对应所有这些可能性和选择性。

另一个更有吸引力的方法，并且也是被 SNMP 所实际采用的方法是利用宏（macro）对在被管对象定义中相互关联的类型进行集合定义。一个宏的定义给出相关类型集合的句法，而宏的实例定义一个特定的类型。因此定义被分为以下等级：

- 宏：定义合法的宏实例，即说明相关集合类型的句法。
- 宏实例：通过为宏定义提供实际参数生成实例，即说明一个特定的类型。
- 宏实例值：用一个特定的值来表示一个特定的实体。

以下是 OBJECT-TYPE 宏的定义（引自 RFC 1212）。

```
OBJECT-TYPE MACRO ::=
        BEGIN
           TYPE NOTATION ::=
                                  -- must conform to
                                  -- RFC1155's ObjectSyntax
                          "SYNTAX" type(ObjectSyntax)
                          "ACCESS" Access
                          "STATUS" Status
                          DescrPart
                          ReferPart
                          IndexPart
                          DefValPart
           VALUE NOTATION ::= value (VALUE ObjectName)

           Access ::= "read-only"
                         | "read-write"
                         | "write-only"
                         | "not-accessible"
           Status ::= "mandatory"
                         | "optional"
                         | "obsolete"
                         | "deprecated"

           DescrPart ::=
                    "DESCRIPTION" value (description DisplayString)
                        | empty

           ReferPart ::=
                    "REFERENCE" value (reference DisplayString)
                        | empty

           IndexPart ::=
                    "INDEX" "{" IndexTypes "}"
                 | empty
           IndexTypes ::=
                    IndexType | IndexTypes "," IndexType
           IndexType ::=
                           -- if indexobject, use the SYNTAX
                           -- value of the correspondent
                           -- OBJECT-TYPE invocation
                    value (indexobject ObjectName)
                           -- otherwise use named SMI type
                           -- must conform to IndexSyntax below
                        | type (indextype)

           DefValPart ::=
                    "DEFVAL" "{" value (defvalue ObjectSyntax) "}"
                        | empty
```

```
END
```

其中的主要项目是：

- SYNTAX：对象类的抽象句法，该句法必须从 SMI 的对象句法类型中确定一种类型。
- ACCESS：定义通过 SNMP 或其他协议访问对象实例的方法。Access 子句定义该对象类型支持的最低等级。可选的等级有 read-only、read-write、write-only 和 not-accessible。
- STATUS：指出该对象在实现上的要求。要求可以是 mandatory（必须）、optional（可选）、deprecated（恳求——必须实现的对象，但很可能在新版 MIB 中被删除）和 obsolete（废除——不再需要被管系统实现的对象）。
- DescrPart：对象类型语义的文本描述。该子句是可选的。
- ReferPart：对定义在其他 MIB 模块中的某个对象的文本型交叉引用。该子句是可选的。
- IndexPart：用于定义表。该子句只是在对象类型对应表中的“行”时才出现。
- DefValPart：定义一个默认值，用于建立对象实例。该子句是可选的。
- VALUE NOTATION：指出通过 SNMP 访问该对象时使用的名字。

由于应用 OBJECT-TYPE 宏的 MIB 的完整的定义包含在 MIB 的冗长的文档中，因此，人们并不常使用它们。比较常用的更简捷的方法——基于树型结构和对象特性的表格表示的方法。

4．定义表格

SMI 只支持一种数据结构化方法，即标量值条目的二维表格。表格的定义用到 ASN.1 的 sequence 和 sequence of 两个类型和 OBJECT-TYPE 宏中的 IndexPart。

表格定义方法可以通过实例进行说明。考虑对象类型 tcpConnTable，这个对象包含由相应的被管实体维护的 TCP connections 的信息。对于每个这样的 connection，以下信息在表中存储：

- state：TCP connection 的状态。
- local address：该 connection 的本端的 IP 地址。
- local port：该 connection 的本端的 TCP 端口。
- remote address：该 connection 的另一端的 IP 地址。
- remote port：该 connection 的另一端的 TCP 端口。

需要注意的是，tcpConnTable 是存放在某个被管系统维护的 MIB 中。因此，tcpConnTable 中的一个条目对应被管系统中的一个 connection 的状态信息。TCP connection 的状态信息有 22 个项目，按照 tcpConnTable 的定义，只有上述 5 个项目对网络管理者来说是可见的。这也体现了 SNMP 强调保持网络管理简单性的特点。即在被管对象中，只包含相对应的被管实体的有限的和有用的信息。

以下给出了 tcpConnTable 的定义（引自 RFC1213）。

```
tcpConnTable OBJECT-TYPE
    SYNTAX  SEQUENCE OF TcpConnEntry
```

```
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
            "A table containing TCP connection-specific
            information."
    ::= { tcp 13 }
tcpConnEntry OBJECT-TYPE
    SYNTAX  TcpConnEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
            "Information about a particular current TCP
            connection.  An object of this type is transient,
            in that it ceases to exist when (or soon after)
            the connection makes the transition to the CLOSED
            state."
    INDEX   { tcpConnLocalAddress,
              tcpConnLocalPort,
              tcpConnRemAddress,
              tcpConnRemPort }
    ::= { tcpConnTable 1 }
TcpConnEntry ::=
    SEQUENCE {
        tcpConnState
            INTEGER,
        tcpConnLocalAddress
            IpAddress,
        tcpConnLocalPort
            INTEGER (0..65535),
        tcpConnRemAddress
            IpAddress,
        tcpConnRemPort
            INTEGER (0..65535)
    }
```

以上可以看到 sequence 和 sequence of 在定义表格时的应用：

- 整个表由一个 SEQUENCE OF TcpConnEntry 构成。ASN.1 的结构 SEQUENCE OF 由一个或多个相同的元素构成，在本例中（在所有的 SNMP SMI 的情况下）每个元素是表中的一行。
- 每一行由一个指定了五个标量元素的 SEQUENCE 构成。ASN.1 的结构 SEQUCECE 由固定数目的元素组成，元素的类型可以是多种。尽管 ASN.1 允许这些元素是可选的，但 SMI 限制这个结构只能使用“mandatory”元素。在本例中，每一行所包含的元素的类型是 INTEGER, IpAddress, INTEGER, IpAddress, INTERGE。

tcpConnEntry 定义中的 INDEX 成分确定哪个对象值将被用于区分表中的各行。在 TCP 中，一个 socket（IP 地址，TCP 端口）可以支持多个 connection，而任意一对 sockets 之间同时只能有一个 connection。因此为了明确的区分各行，每行中的后四个元素是必要的，也是

充分的。

14.2.2 MIB-II

在 TCP/IP 网络管理的建议标准中，提出了多个相互独立的 MIB，其中包含为 Internet 的网络管理而开发的 MIB-II。鉴于它在说明标准 MIB 的结构、作用和定义方法等方面的重要性和代表性，有必要对其进行比较深入的讨论。

MIB-II 是在 MIB-I 的基础之上开发的，是 MIB-I 的一个超集。MIB-II 组被分为以下分组：

- system：关于系统的总体信息。
- interface：系统到子网接口的信息。
- at（address translation）：描述 internet 到 subnet 的地址映射。
- ip：关于系统中 IP 的实现和运行信息。
- icmp：关于系统中 ICMP 的实现和运行信息。
- tcp：关于系统中 TCP 的实现和运行信息。
- udp：关于系统中 UDP 的实现和运行信息。
- egp：关于系统中 EGP 的实现和运行信息。
- dot3（transmission）：有关每个系统接口的传输模式和访问协议的信息。
- snmp：关于系统中 SNMP 的实现和运行信息。

1．system 组

system 组提供有关被管系统的总体信息。表 14.1 列出了该组中各个对象的名称、句法、访问权限和对象描述。

表14.1　system组中的对象

Object	Syntax	Access	Description
sysDescr	DisplayString (SIZE(0 … 255))	RO	对实体的描述，如硬件、操作系统等
sysObjectID	OBJECT IDENTIFIER	RO	实体中包含的网络管理子系统的厂商标识
sysUpTime	TimeTicks	RO	系统的网络管理部分本次启动以来的时间
sysContect	DisplayString (SIZE(0 … 255))	RW	该被管节点负责人的标识和联系信息
sysName	DisplayString (SIZE(0 … 255))	RW	该被管节点被赋予的名称
sysLocation	DisplayString (SIZE(0 … 255))	RW	该节点的物理地点
sysService	INERGER(0 … 127)	RO	指出该节点所提供的服务的集合，7 个 bit 对应 7 层服务

2．interfaces 组

interfaces 组包含实体物理接口的一般信息，包括配置信息和各接口中所发生的事件的统计信息。表 14.2 列出了该组中各个对象的名称、句法、访问权限和对象描述。

表14.2　interfaces组中的对象

Object	Syntax	Access	Description
ifNumber	INTEGER	RO	网络接口的数目
ifTable	SEQUENCE OF ifEntry	NA	接口条目清单
ifEntry	SEQUENCE	NA	包含子网及其以下层对象的接口条目
ifIndex	INTEGER	RO	对应各个接口的惟一值
ifDescr	DisplayString (SIZE(0 … 255))	RO	有关接口的信息，包括厂商、产品名称、硬件接口版本
ifType	INTEGER	RO	接口类型，根据物理或链路层协议区分
ifMtu	INERGER	RO	接口可接收或发送的最大协议数据单元的尺寸
ifSpeed	Gauge	RO	接口当前数据速率的估计值
ifPhysAddress	PhysAddress	RO	网络层之下协议层的接口地址
ifAdminStatus	INTEGER	RW	期望的接口状态（up(1), down(2), testing(3))
ifOperStatus	INTEGER	RO	当前的操作接口状态（up(1), down(2), testing(3))
ifLastChange	TimeTicks	RO	接口进入当前操作状态的时间
ifInOctets	Counter	RO	接口收到的 8 元组的总数
ifInUcastPkts	Counter	RO	递交到高层协议的子网单播的分组数
ifInNUcastPkts	Counter	RO	递交到高层协议的非单播的分组数
ifInDiscards	Counter	RO	被丢弃的进站分组数
ifInErrors	Counter	RO	有错的进站分组数
ifInUnkownProtos	Counter	RO	由于协议未知而被丢弃的分组数
ifOutOctets	Counter	RO	接口发送的 8 元组的总数
ifOutUcastPkts	Counter	RO	发送到子网单播地址的分组总数
ifOutNUcastPkts	Counter	RO	发送到非子网单播地址的分组总数
ifOutDiscards	Counter	RO	被丢弃的出站分组数
ifOutErrors	Counter	RO	不能被发送的有错的分组数
ifOutQLen	Gauge	RO	输出分组队列长度
ifSpecific	OBJECT IDENTIFIER	RO	参考 MIB 对实现接口的媒体的定义

3．address translation 组

address translation 组由一个表构成，表中的每一行对应系统中的一个物理接口，提供网络地址向物理地址的映射。一般情况下，网络地址是指系统在该接口上的 IP 地址，而物理地址决定于实际采用的子网情况。例如，如果接口对应的是 LAN，则物理地址是接口的 MAC 地址，如果对应 X.25 分组交换网，则物理地址可能是一个 X.121 地址。表 14.3 列出了该组中各个对象的名称、句法、访问权限和对象描述。

表14.3　address translation组中的对象

Object	Syntax	Access	Description
atTable	SEQUENCE OF AtEntry	NA	包含网络地址对物理地址的映射
atEntry	SEQUENCE	NA	包含一个网络地址、物理地址对

续表

Object	Syntax	Access	Description
atIfIndex	INTEGER	RW	表格条目的索引
atPhysAddress	PhysAddress	RW	依赖媒体的物理地址
atNetAddress	NetworkAddress	RW	对应物理地址的网络地址

实际上，address translation 组包含在 MIB-II 中只是为了与 MIB-I 兼容，MIB-II 的地址转换信息在各个网络协议组中提供。

4．ip 组

ip 组包含有关节点上 IP 实现和操作的信息，如有关 IP 层流量的一些计数器。ip 组中包含三个表，ipAddrTable、ipRouteTable 和 ipNetToMediaTable。

ipAddrTable 包含分配给该实体的 IP 地址的信息，每个地址被惟一分配给一个物理地址。

ipRouteTable 包含用于互联网路由选择的信息。该路由表中信息是比较原本地从一些协议的路由表中抽取而来的。实体当前所知的每条路由都有一个条目，表格由 ipRouteDest 索引。ipRouteTable 中的信息可用于配置的监测，并且由于表中的对象是 read-write 的，因此也可被用于路由控制。

ipNetToMediaTable 是一个提供 IP 地址和物理地址之间对应关系的地址转换表。除了增加一个指示映射类型的对象 ipNetToMediaType 之外，表中所包含的信息与 address translation 组相同。

此外，ip 组中还包含一些用于性能和故障监测的标量对象。表 14.4 列出了该组中各个对象的名称、句法、访问权限和对象描述。

表14.4　ip组中的对象

Object	Syntax	Access	Description
ipForwarding	INTEGER	RW	是否作为 IP 网关（1/0）
ipDefaultTTL	INTEGER	RW	插入到该实体生成的数据报的 IP 头中 Time-To-Live 字段中的默认值
ipInReceives	Counter	RO	接口收到的输入数据报的总数
ipInHdrErrors	Counter	RO	由于 IP 头错被丢弃的输入数据报总数
ipInAddrErrors	Counter	RO	由于 IP 地址错被丢弃的输入数据报总数
ipForwDatagrams	Counter	RO	转发的输入数据报数
ipInUnknownProtos	Counter	RO	由于协议未知被丢弃的输入数据报数
ipInDiscards	Counter	RO	无适当理由而被丢弃的输入数据报数
ipInDelivers	Counter	RO	成功地递交给 IP 用户协议的输入数据报数
ipOutRequests	Counter	RO	本地 IP 用户协议要求传输的 IP 数据报总数
ipOutNoRoutes	Counter	RO	由于未找到路由而被丢弃的 IP 数据报数
ipReasmTimeOut	INTEGER	RO	重组接收到的碎片可等待的最大秒数
ipReasmReqds	Counter	RO	接收到的需要重组的 IP 碎片数
ipReasmOKs	Counter	RO	成功重组的 IP 数据报数
ipRaesmFails	Counter	RO	由 IP 重组算法检测到的重组失败的数目
ipFragsOk	Counter	RO	成功拆分的 IP 数据报数

续表

Object	Syntax	Access	Description
ipFragsFails	Counter	RO	不能成功拆分而被丢弃的 IP 数据报数
ipFragsCreates	Counter	RO	本实体产生的 IP 数据报碎片数
ipAddrTable	SEQUENCE OF IpAddrEntry	NA	本实体的 IP 地址信息（表内对象略）
ipRouteTable	SEQUENCE OF I pRouteEntry	NA	IP 路由表（表内对象略）
ipNetToMediaTable	SEQUENCE OF IpNetToMedis Entry	NA	用于将 IP 映射到物理地址的地址转换表（表内对象略）
IpRouting Discards	Counter	RO	被丢弃的路由选择条目

5．icmp 组

ICMP（Internet Control Message Protocol）是 TCP/IP 协议族中的一部分，所有实现 IP 协议的系统都提供 ICMP。ICMP 提供从路由器或其他主机向主机传递消息的手段，它的基本作用是反馈通信环境中存在的问题，例如：数据报不能到达目的地，路由器没有缓冲区容量来转发数据报。

icmp 组包含有关一个节点的 ICMP 的实现和操作的信息，具体地讲，icmp 组由节点接收和发送的各种 ICMP 消息的计数器所构成的。表 14.5 列出了该组中各个对象的名称、句法、访问权限和对象描述。

表14.5　icmp组中的对象

Object	Syntax	Access	Description
icmpInMsgs	Counter	RO	收到的 ICMP 消息的总数
icmpInErrors	Counter	RO	收到的有错的 ICMP 的消息数
icmpInDestUnreachs	Counter	RO	收到的目的地不可到达的消息数
icmpInTimeExcds	Counter	RO	收到的超时的消息数
icmpInParmProbs	Counter	RO	收到的有参数问题的消息数
icmpInSrcQuenchs	Counter	RO	收到的源有问题的消息数
icmpInRedirects	Counter	RO	收到的重定向的消息数
icmpInEchos	Counter	RO	收到的要求 echo 的消息数
icmpInEchoReps	Counter	RO	收到的应答 echo 的消息数
icmpInTimestamps	Counter	RO	收到的要求 Timestamp 的消息数
icmpInTimestampReps	Counter	RO	收到的应答 Timestamp 的消息数
icmpInAddrMasks	Counter	RO	收到的要求 Address Mask 的消息数
icmpInAddrMaskReps	Counter	RO	收到的应答 Address Mask 的消息数
icmpOutMsgs	Counter	RO	发出的 ICMP 消息的总数
icmpOutErrors	Counter	RO	发出的有错的 ICMP 的消息数
icmpOutDestUnreachs	Counter	RO	发出的目的地不可到达的消息数
icmpOutTimeExcds	Counter	RO	发出的超时的消息数
icmpOutParmProbs	Counter	RO	发出的有参数问题的消息数
icmpOutSrcQuenchs	Counter	RO	发出的源有问题的消息数

续表

Object	Syntax	Access	Description
icmpOutRedirects	Counter	RO	发出的重定向的消息数
icmpOutEchos	Counter	RO	发出的要求 echo 的消息数
icmpOutEchoReps	Counter	RO	发出的应答 echo 的消息数
icmpOutTimestamps	Counter	RO	发出的要求 Timestamp 的消息数
icmpOutTimestampReps	Counter	RO	发出的应答 Timestamp 的消息数
icmpOutAddrMasks	Counter	RO	发出的要求 Address Mask 的消息数
icmpOutAddrMaskReps	Counter	RO	发出的应答 Address Mask 的消息数

6．tcp 组

tcp 组包含有关一个节点的 TCP 的实现和操作的信息，以上定义的 tcpConnTable 包含在这个组中。表 14.6 列出了该组中各个对象的名称、句法、访问权限和对象描述。

表14.6　tcp组中的对象

Object	Syntax	Access	Description
tcpRtoAlgorithm	INTEGER	RO	重传时间
tcpRtoMin	INTEGER	RO	重传时间的最小值
tcpRtoMax	INTEGER	RO	重传时间的最大值
tcpMaxConn	INTEGER	RO	实体支持的 TCP 连接数的上限
tcpActiveOpens	Counter	RO	实体已经支持的主动打开的数量
tcpPassiveOpens	Counter	RO	实体已经支持的被动打开的数量
tcpAttemptFails	Counter	RO	已经发生的试连失败的次数
tcpEstabResets	Counter	RO	已经发生的复位的次数
tcpCurrEstab	Gauge	RO	当前状态为 established 的 TCP 连接数
tcpInSegs	Counter	RO	收到的 segments 总数
tcpOutSegs	Counter	RO	发出的 segments 总数
tcpRetranSegs	Counter	RO	重传的 segments 总数
tcpConnTable	SEQUENCE OF TcpConnEntry	NA	包含 TCP 各个连接的信息（表内对象略，参考 tcp ConnTable 的定义）
tcpInErrors	Counter	RO	收到的有错的 segments 的总数
tcpOutRsts	Counter	RO	发出的含有 RST 标志的 segments 数

7．udp 组

udp 组包含有关一个节点的 UDP 的实现和操作的信息。除了有关发送和接收的数据报的信息之外，这个组中还包含一个 udpTable 表，该表中包含 UDP 端点的管理信息。所谓 UDP 端点是指正在支持本地应用接收数据报的 UDP 进程。udpTable 表中包含每个 UDP 端点用户的 IP 地址和 UDP 端口。表 14.7 列出了该组中各个对象的名称、句法、访问权限和对象描述。

表14.7　udp组中的对象

Object	Syntax	Access	Description
udpInDatagrams	Counter	RO	递交该 UDP 用户的数据报的总数

续表

Object	Syntax	Access	Description
udpNoPorts	Counter	RO	收到的目的端口上没有应用的数据报总数
udpInErrors	Counter	RO	收到的无法递交的数据报数
udpOutDatagrams	Counter	RO	该实体发出的 UDP 数据报总数
udpTable	SEQUENCE OF UdpEntry	NA	包含 UDP 的用户信息
udpTable	SEQUENCE	NA	某个当前 UDP 用户的信息
udpLocalAddress	IpAddress	RO	UDP 用户的本地 IP 地址
udpLocalPort	INTEGER	RO	UDP 用户的本地端口号

8．egp 组

egp 组包含有关一个节点的 EGP（External Gateway Protocol）的实现和操作的信息。除了有关发送和接收的 EGP 消息的信息之外，这个组中还包含一个 egpNeighTable 表，该表中包含有关相邻网关的信息。表 14.8 列出了该组中各个对象的名称、句法、访问权限和对象描述。

表14.8　egp组中的对象

Object	Syntax	Access	Description
egpInMsgs	Counter	RO	收到的无错的 EGP 消息数
egpInErrors	Counter	RO	收到的有错的 EGP 消息数
egpOutMsgs	Counter	RO	本地产生的 EGP 消息总数
egpOutErrors	Counter	RO	由于资源限制没有发出的本地产生的 EGP 消息数
egpNeighTable	SEQUENCE OF EgpNeighEntry	NA	相邻网关的 EGP 表（表内的对象略）
egpAs	INTEGER	RO	本 EGP 实体的自治系统数

14.3　简单网络管理协议

14.3.1　SNMP 支持的操作

SNMP 只支持对变量的检查和修改的操作，具体地可以对标量对象进行以下 3 种操作。

- Get：管理站从被管理站提取标量对象值。
- Set：管理站更新被管理站中的标量对象值。
- Trap：被管理站向管理站主动地发送一个标量对象值。

MIB 的结构不能通过增加或减少对象实例被改变，并且，访问只能对对象标识树中的叶子对象进行。这些限制大大简化了 SNMP 的实现，但同时也限制了网络管理系统的能力。

14.3.2 共同体和安全控制

网络管理是一种分布式的应用。与其他分布式的应用相同，网络管理中包含由一个应用协议支持的多个应用实体的相互作用。在 SNMP 网络管理中，这些应用实体就是采用 SNMP 的管理站应用实体和被管理站的应用实体。

SNMP 网络管理具有一些不同于其他分布式应用的特性，它包含一个管理站和多个被管理站之间一对多的关系。即管理站能够获取和设置各管理站的对象，能够从各被管理站中接收陷阱信息。因此，从操作或控制的角度来看，管理站管理着多个被管理站。同时，系统中也可能有多个管理站，每个管理站都管理所有的或一部分被管理站。

反过来，我们也要看到 SNMP 网络管理中还包含另外一种一对多的关系——一个被管理站和多个管理站之间的关系。每个被管理站控制着自己的本地 MIB，同时必须能够控制多个管理站对这个本地 MIB 的访问。这里所说的控制有以下三个方面：

- ❐ 认证服务：将对 MIB 的访问限定在授权的管理站的范围内。
- ❐ 访问策略：对不同的管理站给予不同的访问权限。
- ❐ 代管服务：一个被管理站可以作为其他一些被管理站（托管站）的代管，这就要求在这个代管系统中实现为托管站服务的认证服务和访问权限。

以上这些控制都是为了保证网络管理信息的安全，即被管系统需要保护它们的 MIB 不被非法地访问。SNMP 通过共同体（community）的概念提供了初步的和有限的安全能力。

SNMP 用共同体来定义一个代理者和一组管理者之间的认证、访问控制和代管的关系。共同体是一个在被管系统中定义的本地的概念。被管系统为每组可选的认证、访问控制和代管特性建立一个共同体。每个共同体被赋予一个在被管系统内部惟一的共同体名，该共同体名要提供给共同体内的所有的管理站，以便它们在 get 和 set 操作中应用。代理者可以与多个管理站建立多个共同体，同一个管理站可以出现在不同的共同体中。

由于共同体是在代理者处本地定义的，因此不同的代理者处可能会定义相同的共同体名。共同体名相同并不意味着共同体有什么相似之处，因此，管理站必须将共同体名与代理者联系起来加以应用。

1．认证服务

认证服务是为了保证通信是可信的。在 SNMP 消息的情况下，认证服务的功能是保证收到的消息是来自它所声称的消息源。SNMP 只提供一种简单的认证模式：所有由管理站发向代理者的消息都包含一个共同体名，这个名字发挥口令的作用。如果发送者知道这个口令，则认为消息是可信的。

通过这种有限的认证形式，网络管理者可以对网络监控（set、trap）特别是网络控制（set）操作进行限制。共同体名被用于引发一个认证过程，而认证过程可以包含加密和解密以实现更安全的认证。

2．访问策略

通过定义共同体，代理者将对它的 MIB 的访问限定在了一组被选择的管理站中。通过使用多个共同体，代理者可以为不同的管理站提供不同的 MIB 访问控制。访问控制包含两个方面：

- ❐ SNMP MIB 视图：MIB 中对象的一个子集。可以为每个共同体定义不同的 MIB 视

图，视图中的对象子集可以不在 MIB 的一个子树之内。

- SNMP 访问模式：READ-ONLY 或 READ-WRITE，为每个共同体定义一个访问模式。

MIB 视图和访问模式的结合被称为 SNMP 共同体轮廓（profile）。即，一个共同体轮廓由代理者处 MIB 的一个子集加上一个访问模式构成。SNMP 访问模式统一地被用于 MIB 视图中的所有对象。因此，如果选择了 READ-ONLY 访问模式，则管理站对视图中的所有对象都只能进行 read-only 操作。

事实上，在一个共同体轮廓之内，存在两个独立的访问限制——MIB 对象定义中的访问限制和 SNMP 访问模式。这两个访问限制在实际应用中必须得到协调。表 14.9 给出了这两个访问限制的协调规则。注意，对象被定义为 write-only，SNMP 也可以对其进行 read 操作。

表14.9　MIB对象定义中的ACCESS限制与SNMP访问模式的关系

MIB 对象定义中的 ACCESS 限制	SNMP 访问模式	
	READ-ONLY	READ-WRITE
read-only	get 和 trap 操作有效	
read-write	get 和 trap 操作有效	get，set 和 trap 操作有效
Write-only	get 和 trap 操作有效，但操作值与具体实现有关	get，set 和 trap 操作有效，但操作值与具体实现有关
not-accessible	无效	

在实际应用中，一个共同体轮廓要与代理者定义的某个共同体联系起来，便构成了 SNMP 的访问策略（access policy）。即 SNMP 的访问策略指出一个共同体中的 MIB 视图及其访问模式。

3．代管服务

共同体的概念对支持代管服务也是有用的。如前所述，在 SNMP 中，代管是指为其他设备提供管理通信服务的代理者。对于每个托管设备，代管系统维护一个对它的访问策略，以此使代管系统知道哪些 MIB 对象可以被用于管理托管设备和能够用何种模式对它们进行访问。

14.3.3　实例标识

我们已经看到，MIB 中的每个对象都有一个由其在树型结构的 MIB 中所处的位置所定义的惟一的对象标识符。但是，应该注意到，MIB 树型结构给出的对象标识符在一些情况下只是对象类型的标识符，不能惟一地标识对象的实例。例如表格的对象标识符不能标识表格中各个条目。由于对 MIB 的访问是对对象实例的访问，因此各个对象实例都必须有惟一标识的方法。

1．纵列对象

表中的对象被称为纵列对象。纵列对象标识符不能独自标识对象实例，因为表中的每一行都有纵列对象的一个实例。为了实现这类对象实例的惟一标识，SNMP 实际定义了两种技术：顺序访问技术和随机访问技术。顺序访问技术是通过利用辞典编排顺序实现的。而随机

访问技术是通过利用索引对象值实现的。下面首先讨论随机访问技术。

一个表格是由零到多个行（条目）构成的，每一行都包含一组相同的标量对象类型，或称纵列对象。每个纵列对象都有一个惟一的标识符。但由于纵列对象可能有多个实例，因此纵列对象标识符并不能惟一标识它的各个实例。然而，在定义表格时，一般包含一个特殊的纵列对象 INDEX，即索引对象，它的每个实例都具有不同的值，可以用来标识表中的各行。因此，SNMP 采用将索引对象值连接在纵列对象标识符之后的方法来标识纵列对象的实例。

作为例子，我们看一下 interfaces 组中的 ifTable。表中有一个索引对象 ifIndex，它的值是一个 1 到 ifNumber 之间的整数，对应每个接口，ifIndex 有一个惟一的值。现在假设要获取系统中第 2 个接口的接口类型 ifType。ifType 的对象标识符是 1.3.6.1.2.1.2.2.1.3。而第 2 个接口的 ifIndex 值是 2。因此对应第 2 个接口的 ifType 的实例的标识符便为 1.3.6.1.2.1.2.2.1.3.2。即将这个 ifIndex 的值作为实例标识符的最后一个子标识符加到 ifType 对象标识符之后。

2．表格及行对象

对于表格和行对象，没有定义它们的实例标识符。这是因为表格和行不是叶子对象，因而不能由 SNMP 访问。在这些对象的 MIB 定义中，它们的 ACCESS 特性被设为 not-accessible。

3．标量对象

在标量对象的场合，用对象类型标识符便能惟一标识它的实例，因为每个标量对象类型只有一个对象实例。但是，为了与表格对象实例标识符的约定保持一致，也为了区分对象的类型和对象实例，SNMP 规定标量对象实例的标识符由其对象类型标识符加 0 组成。

14.3.4 辞典编纂式排序

对象标识符是反映该对象在 MIB 中的树型结构的一个整数序列。给出一个 MIB 的树型结构，跟踪从 root 开始到某个特定对象的路径，便可以得到该对象的对象标识符。

由于对象标识符是一个整数序列，因此，可以把它们看作是某本书的内容在书中的章节排序。总排序可以通过遍历 MIB 中的对象标识符树来生成。利用这个总排序，也可以对对象实例进行惟一的标识。

因为网络管理站对代理者提供 MIB 视图的构成不一定完全清楚，因此，它需要一种不必提供对象名称而能访问对象的方法。在这种情况下，对象及其实例的排序就是非常重要的。利用这个排序，管理站可以有效地遍历一个 MIB 的结构。因为管理站只要提供树型结构的任意一点上的一个对象实例的标识符，就可以顺序地对其后继的对象实例进行访问。

14.3.5 SNMP 消息格式

管理站和代理者之间以传送 SNMP 消息的形式交换信息。每个消息包含一个指示 SNMP 版本号的版本号，一个用于本次交换的共同体名和一个指出 5 种协议数据单元之一的消息类型。图 14.4 描述了这种结构。表 14.10 对其中的元素进行了说明。

1．SNMP 消息的发送

一般情况下，一个 SNMP 协议实体完成以下动作向其他 SNMP 实体发送 PDU：

❒ 构成 PDU。

- ❒ 将构成的 PDU、源和目的传送地址以及一个共同体名传给认证服务。认证服务完成所要求的变换，例如进行加密或加入认证码，然后将结果返回。
- ❒ SNMP 协议实体将版本字段、共同体名以及上一步的结果组合成为一个消息。
- ❒ 用基本编码规则（BER）对这个新的 ASN.1 的对象编码，然后传给传输服务。

Version	Community	SNMP PDU

（a）GetRequest PDU, GetNextRequest PDU, SetRequest PDU

PDU type	request-id	O	O	variable-bindings

（b）GetRequest-PDU, GetNextRequest-PDU, SetRequest-PDU

PDU type	request-id	error-status	error-index	variable-bindings

（c）Response PDU

PDU type	enterprise	agent-addr	generic-trap	specific-trap	timp-stamp	variable-bindings

（d）Trap PDU

name1	value1	name2	value2	. . .	name n	value n

（e）Variable-bindings

图 14.4　SNMP 消息格式

表14.10　SNMP消息字段

字　　段	描　　述
version	SNMP 版本
community	共同体的名字用作 SNMP 认证消息的口令
request-id	为每个请求赋予一个惟一的标识符
error-status	noError(0),tooBig(1),noSuchName(2),badValue(3),readOnly(4),genErr(5)
error-index	当 error-status 非 0 时，可以进一步提供信息指出哪个变量引起的问题
variable-bindings	变量名及其对应值清单
enterprise	生成 trap 的对象的类型
agent-addr	生成 trap 的对象的地址
generic-trap	一般的 trap 类型 :coldStart(0),warmStart(1),linkDown(2),linkUp(3), authentication -Failure(4), egpNeighborLoss(5),enterprise-Specific(6)
specific-trap	特定的 trap 代码
time-stamp	网络实体从上次启动到本 trap 生成所经历的时间

2．SNMP 消息的接收

一般情况下，一个 SNMP 协议实体完成以下动作接收一个 SNMP 消息。



- ❒ 进行消息的基本句法检查，丢弃非法消息。
- ❒ 检查版本号，丢弃版本号不匹配的消息。
- ❒ SNMP 协议实体将用户名、消息的 PDU 部分以及源和目的传输地址传给认证服务。如果认证失败，认证服务通知 SNMP 协议实体，由它产生一个 trap 并丢弃这个消息，如果认证成功，认证服务返回 SNMP 格式的 PDU。
- ❒ 协议实体进行 PDU 的基本句法检查，如果非法，丢弃该 PDU，否则利用共同体名选择对应的 SNMP 访问策略，对 PDU 进行相应处理。

3．变量绑定

在 SNMP 中，可以将多个同类操作（get、set、trap）放在一个消息中。如果管理站希望得到一个代理者处的一组标量对象的值，它可以发送一个消息请求所有的值，并通过获取一个应答得到所有的值。这样可以大大减少网络管理的通信负担。

为了实现多对象交换，所有的 SNMP 的 PDU 都包含了一个变量绑定字段。这个字段由对象实例的一个参考序列及这些对象的值构成。某些 PDU 只需给出对象实例的名字，如 get 操作。对于这样的 PDU，接收协议实体将忽略变量绑定字段中的值。

14.3.6　GetRequest PDU

SNMP 实体应网络管理站应用程序的请求发出 GetRequest PDU。发送实体将以下字段包含在 PDU 之中：

- ❒ PDU 类型：指出 GetRequest PDU 类型。
- ❒ request-id：Request-id 能够使 SNMP 应用将得到的各个应答与发出的各个请求一一对应起来。同时也可以使 SNMP 实体能够处理由于传输服务的问题而产生的重复的 PDU。
- ❒ variablebindings：要求获取值的对象实例清单。

GetRequest PDU 的 SNMP 接收实体用包含相同 request-id 的 GetResponse PDU 进行应答。GetRequest 操作是原子操作——要么所有的值都提取回来，要么一个都不提取。

GetRequst 操作不成功的原因有对象名不匹配（noSuchName）、返回结果太长（tooBig）以及其他原因（genErr）。

SNMP 只允许提取 MIB 树中的叶子对象的值。因此不能只提供一个表或一个条目的名字来获取整个表或整行的对象值。但是可以将表中每行的各个对象包含在变量绑定中，来一次获取一行的对象值。

14.3.7　GetNextRequest PDU

GetNextRequest PDU 几乎与 GetRequest PDU 相同。它们具有相同交换模式和相同的格式。惟一的不同是：在 GetRequest PDU 中，变量绑定字段中列出的是要取值的对象实例名本身，而在 GetNextRequest PDU 中，变量绑定字段列出的是要取值的对象实例的“前一个”对象实例名。与 GetRequest 相同，GetNextRequest 也是原子操作。

虽然与 GetRequest 的外在差异不大，但是 GetNextRequest 却有 GetRequest 无法替代的用途。它能够使网络管理站去动态地发现一个 MIB 视图的结构。它也为查找不知其条目的表

提供了一个有效的机制。

1．简单对象值的提取

假设网络管理站希望从某个代理者处提取 udp 组中的所有简单对象，则它可以发出一个如下的 PDU：

GetRequest(udpInDatagrams.0，udpNoPorts.0,udpInError.0,udpOutDatagrams.0)

如果代理者支持所有这些对象，则将返回一个包含这 4 个对象值的 GetResponse PDU：

GetResponse((udpInDatagrams.0 = 100), (udpNoPorts.0 = 1), (udpInErrors.0 = 2),
(udpOutDatagrams.0 = 200))

这里，100，1，2 和 200 分别是这 4 个对象的值。然而，只要有一个对象不被支持，则代理者将返回一个含有错误码 NoSuchName 的 GetResponse PDU，而不返回任何其他值。为了确保得到所有可用的对象值，管理站必须分别发出 4 个 GetRequest PDU。

现在考虑应用 GetNextRequest PDU 的情况：

GetNextRequest (udpInDatagrams, udpNoPorts, udpInErrors, udpOutDatagrams)

其中，udpInDatagrams = 1.3.6.1.2.1.7.1，udpNoPorts = 1.3.6.1.2.1.7.2，udpInErrors = 1.3.6.1.2.1.7.3，udpOutDatagrams = 1.3.6.1.2.1.7.4。

在这种情况下，代理者将返回清单中每个标识符的“下一个”对象实例的值。假设 4 个对象都被支持，则代理者返回一个如下的 GetResponse PDU：

GetResponse((udpInDatagrams.0 = 100), (udpNoPorts.0 = 1), (udpInErrors.0 = 2),
(udpOutDatagrams.0 = 200))

这与前面的情况相同。假设 udpNoPorts 在本视图中是不存在（不可见）的，则代理者的应答为：

GetResponse((udpInDatagrams.0 = 100), (udpInErrors.0 = 2), (udpInErrors.0 = 2),
(udpOutDatagrams.0 = 200))

由于 udpNoPorts.0 = 1.3.6.1.2.1.7.2.0 在本 MIB 视图中是不存在的标识符，因此 udpNoPorts 的“下一个”对象实例便成了 udpInError.0 = 1.3.6.1.2.1.7.3.0。

通过对比可知，GetNextRequest 在提取一组对象值时比 GetRequest 效率更高，更灵活。

2．提取未知对象

GetNextRequest 要求代理者提取所提供的对象标识符的下一个对象实例的值，因此，发送这类 PDU 时，并不要求提供 MIB 视图中实际存在的对象或对象实例的标识符。利用这一特点，管理站可以使用 GetNextRequest PDU 去探查一个 MIB 视图，并搞清它的结构。在上面的例子中，如果管理站发出一个 GetNextRequest（udp）PDU，则将获得 Response（udpInDatagrams.0 = 100）的应答。管理站因此便知道了在这个 MIB 视图中第一个被支持的对象是 udpInDatagrams，并且知道了它的当前值。

14.3.8　SetRequest PDU

SNMP 实体应网络管理站应用程序的请求发出 SetRequest PDU。它与 GetRequest PDU 具有相同的交换模式和相同的格式。但是，SetRequest 是被用于写对象值而不是读。因而，变量绑定清单中既包含对象实例标识符，也包含每个对象实例将被赋予的值。

SetRequest PDU 的 SNMP 接收实体用包含相同 request-id 的 GetResponse PDU 进行应答。

SetRequest 操作是原子操作——要么变量绑定中的所有变量都被更新，要么一个都不被更新。如果应答实体能够更新变量绑定中的所有变量，则 GetResponse PDU 中包含提供给各个变量的值的变量绑定字段。只要有一个变量值不能成功地设置，则无变量值返回，也无变量值被更新。在 GetRequest 操作中可能返回的错误——noSuchName、tooBig 和 genErr 也是 SetRequest 可能返回的错误。另外一个可能返回的错误是 badValue，只要 SetRequest 中有一个变量名和变量值不一致的问题，就会返回这个错误。所谓不一致可能是类型的问题，也可能是长度的问题，还可能是提供的实际的值有问题。

利用 SetRequest 不仅可以对叶子对象实例进行值的更新，也可以利用变量绑定字段进行表格的行增加和行删除操作。

除此之外，SetRequest 还可被用于完成某种动作。SNMP 没有提供一种命令代理者完成某种动作的机制，它的全部能力就是在一个 MIB 视图内 get 和 set 对象值。但是利用 set 的功能可以间接地发布完成某种动作的命令。某个对象可以代表某个命令，当它被设置为特定值时，就执行特定的动作。例如代理者可以设一个初始值为 0 的对象 reBoot，如果管理站将这个对象值置 1，则代理者系统被重新启动，reBoot 的值也被重新置 0。

14.3.9　Trap PDU

SNMP 实体应网络管理代理者应用程序的请求发出 Trap PDU。它被用于向管理站异步地通报某个重要事件。它的格式与其他的 SNMP PDU 完全不同。所包含的字段有：

- PDU 类型：指出 Trap PDU 类型。
- Enterprise：标识产生本 Trap 的网络管理子系统（用 System 组中的 sysObjectId 值）。
- agent-addr：产生本 Trap 的对象的 IP 地址。
- generic-trap：一种预定义的 trap。
- specific-trap：更明确地指出 trap 特性的代码。
- time-stamp：发出 trap 的网络实体从上次重启到产生本 trap 所经历的时间。
- variablebindings：有关 trap 的附加信息（本字段的意义与具体实现有关）。

14.3.10　传输层的支持

SNMP 需要利用传输层的服务来传递 SNMP 消息，但是它并未假定传输层的服务是可靠的还是非可靠的，是无连接的还是面向连接的。

实际上，在 TCP/IP 体系中，SNMP 的实现几乎都是使用无连接协议用户数据报（UDP）。UDP 头中包含源和目的端口字段，允许应用层协议，如 SNMP 填写地址。它还包含一个可选的覆盖 UDP 头和用户数据的校验和（checksum）。如果校验和有问题，UDP 片段（segment）被丢弃。两个端口号给 SNMP 应用，用于代理者侦听 GetRequest, GetNextRequest 和 SetRequest 命令的 161 端口和用于管理站侦听 Trap 命令的 162 端口。

由于 UDP 是非可靠的，因此 SNMP 的消息可能被丢失。SNMP 本身也不保证消息的可靠传递，因此，处理消息丢失问题的负担只能由 SNMP 的用户自己承担。

如何处理 SNMP 消息的丢失没有标准的方法，只能凭通常的感觉处理。在 GetRequest 和 GetNextRequest 的场合，如果在规定的时间内得不到应答，管理站可以认为或者是发出的

命令消息被丢失，或者是代理者返回的应答被丢失。管理站可以再次或多次重发请求，直至成功或最终放弃。由于相同的请求具有相同的 request-id，因此重发可能会使接收者收到多个相同的消息，但这并不会引起问题，因为接收者可以简单地将收到的重复的消息丢弃。

在 SetRequest 的场合，如果在规定的时间内得不到应答，为了确认操作是否成功，可以用 GetRequest 操作进行确认。如果确认 set 操作没被执行，可以重发 SetRequest。

由于 SNMP 的 Trap 没有应答消息，因此没有简单的方法去检验 Trap 的传递。在 SNMP 中，Trap 一般用于提供重要事件的早期告警，作为后备方法，管理站还要定期地轮询代理者获取相关的状态。

14.4　SNMPv2

14.4.1　SNMPv2 对 SNMPv1 的改进

1993 年，SNMP 的改进版 SNMPv2 开始发布，从此，原来的 SNMP 便被称为 SNMPv1。最初的 SNMPv2 最大的特色是增加了安全特性，因此被称为安全版 SNMPv2。但不幸的是，经过几年试用，没有得到厂商和用户的积极响应，并且也发现自身还存在一些严重缺陷。因此，在 1996 年正式发布的 SNMPv2 中，安全特性被删除。这样，SNMPv2 对 SNMPv1 的改进程度便受到了很大的削弱。

总的来说，SNMPv2 的改进主要有以下 3 个方面：

- 支持分布式管理。
- 改进了管理信息结构。
- 增强了管理信息通信协议的能力。

SNMPv1 采用的是集中式网络管理模式。网络管理站的角色由一个主机担当。其他设备（包括代理者软件和 MIB）都由管理站监控。随着网络规模和业务负荷的增加，这种集中式的系统已经不再适应需要。管理站的负担太重，并且来自各个代理者的报告在网上产生大量的业务量。而 SNMPv2 不仅可以采用集中式的模式，也可以采用分布式模式。在分布式模式下，可以有多个顶层管理站，被称为管理服务器。每个管理服务器可以直接管理代理者。同时，管理服务器也可以委托中间管理者担当管理者角色监控一部分代理者。对于管理服务器，中间管理器又以代理者的身份提供信息和接受控制。这种体系结构分散了处理负担，减小了网络的业务量。

SNMPv2 的管理信息结构（SMI）在几个方面对 SNMPv1 的 SMI 进行了扩充。定义对象的宏中包含了一些新的数据类型。最引人注目的变化是提供了对表中的行进行删除或建立操作的规范。新定义的 SNMPv2 MIB 包含有关 SNMPv2 协议操作的基本流量信息和有关 SNMPv2 管理者和代理者的配置信息。

在通信协议操作方面，最引人注目的变化是增加了两个新的 PDU——GetBulkRequest 和 InformRequest。前者使管理者能够有效地提取大块的数据，后者使管理者能够向其他管理者发送 trap 信息。

14.4.2 SNMPv2 网络管理框架

SNMPv2 提供了一个建立网络管理系统的框架。但网络管理应用，如故障管理、性能监测、计费等不包括在 SNMPv2 的范围内。用术语来说，SNMPv2 提供的是网络管理基础结构。图 14.5 是这种基础结构的一个配置实例。

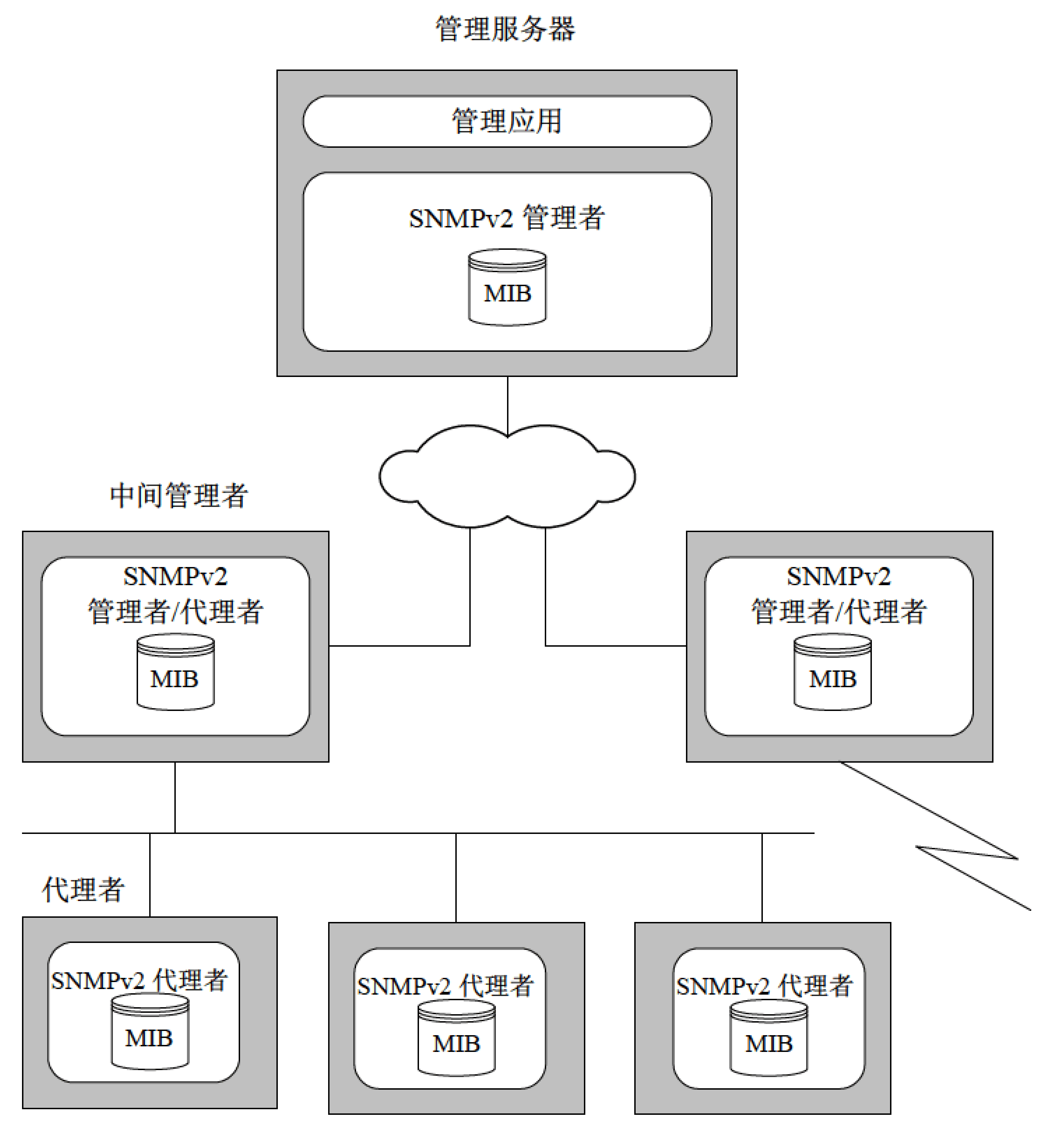


图 14.5　SNMPv2 的配置

SNMPv2 本质上是一个交换管理信息的协议。网络管理系统中的每个角色都维护一个与网络管理有关的 MIB。SNMPv2 的 SMI 对这些 MIB 的信息结构和数据类型进行定义。SNMPv2 提供了一些一般的通用的 MIB，厂商或用户也可以定义自己私有的 MIB。

在配置中至少有一个系统负责整个网络的管理。这个系统就是网络管理应用驻留的地方。管理站可以设置多个，以便提供冗余或分担大网络的管理责任。其他系统担任代理者角色。代理者收集本地信息并保存，以备管理者提取。这些信息包括系统自身的数据，也可以包括网络的业务量信息。

SNMPv2 既支持高度集中化的网络管理模式，也支持分布式的网络管理模式。在分布式模式下，一些系统担任管理者和代理者两种角色，这种系统被称为中间管理者。中间管理者以代理者身份从上级管理系统接受管理信息操作命令，如果这些命令所涉及的管理信息在本

地 MIB 中，则中间管理者便以代理者身份进行操作并进行应答，如果所涉及的管理信息在中间管理者的下属代理者的 MIB 中，则中间管理者先以管理者身份对下属代理者进行发布操作命令，接收应答，然后再以代理者身份向上级管理者应答。

所有这些信息交换都利用 SNMPv2 通信协议实现。与 SNMPv1 相同，SNMPv2 协议仍是一个简单的请求（request）/应答（response）型协议，但在 PDU 种类和协议功能方面对 SNMPv1 进行了扩充。

14.4.3　协议操作

1．SNMPv2 消息

与 SNMPv1 相同，SNMPv2 以包含协议数据单元（PDU）的消息的形式交换信息。外部的消息结构中包含一个用于认证的共同体名。

SNMPv2 确定的消息结构如下：

```
Message :: = SEQUENCE {
        version     INTEGER { version (1) },    -- SNMPv2的版本号为1
        community   OCTET STRING,               -- 共同体名
        data        ANY                         -- SNMPv2 PDU
        }
```

14.3.2 节中对于共同体名、共同体轮廓和访问策略的讨论同样适用于 SNMPv2。

SNMPv2 消息的发送和接收过程与 14.3.5 节中描述的 SNMPv1 消息的发送和接收过程相同。

2．PDU 格式

在 SNMPv2 消息中可以传送 7 类 PDU。表 14.11 列出了这些 PDU，同时指出了对 SNMPv1 也有效的 PDU。图 14.6 描述了 SNMPv2 PDU 的一般格式。

表14.11　SNMP协议数据单元（PDUs）

PDU	描　　述	SNMPv1	SNMPv2
Get	管理者通过代理者获得每个对象的值	●	●
GetNext	管理者通过代理者获得每个对象的下一个值	●	●
GetBulk	管理者通过代理者获得每个对象的 N 个值		●
Set	管理者通过代理者为每个对象设置值	●	●
Trap	代理者向管理者传送随机信息	●	●
Inform	管理者向代理者传送随机信息		●
Response	代理者对管理者的请求进行应答	●	●

值得注意的是，GetRequest、GetNextRequest、SetRequest、SNMPv2-Trap、InformReques 五种 PDU 具有完全相同的格式，并且也可以看作是 error-status 和 error-index 两个字段被置零的 Response PDU 的格式。这样设计的目的是为了减少 SNMPv2 实体需要处理的 PDU 格式种类。

（1）GetRequest PDU

SNMPv2 的 GetRequest PDU 的语法和语义都与 SNMPv1 的 GetRequest PDU 相同，差别是对应答的处理。SNMPv1 的 GetRequest 是原子操作：要么所有的值都返回，要么一个也不

返回，而 SNMPv2 能够部分地对 GetRequest 操作进行应答。即使有些变量值提供不出来，变量绑定字段也要包含在应答的 GetResponse PDU 之中。如果某个变量有意外情况（noSuchObject, noSuchInstance,endOfMibView），则在变量绑定字段中，这个变量名与一个代表意外情况的错误代码而不是变量值配对。

PDU type	request-id	0	0	variable-bindings

（a）GetRequest-PDU, GetNextRequest-PDU, SetRequest-PDU, SNMPv2-Trap-PDU,InformRequest-PDU

PDU type	request-id	error-status	error-index	variable-bindings

（b）Response-PDU

PDU type	request-id	non-repeaters	max-repetitions	variable-bindings

（c）GetBulkRequest-PDU

name1	value1	name2	value2	. . .	name n	value n

（d）Variable-bindings

图 14.6 SNMPv2 PDU 格式

在 SNMPv2 中，按照以下规则处理 GetRequest 变量绑定字段中的每个变量来构造应答 PDU：

- 如果 OBJECT IDENTIFIER 前缀与该请求在代理者处所能访问的变量的前缀都不匹配，则它的值字段被设置为 noSuchObject。
- 否则，如果变量名与该请求在代理者处所能访问的变量的名称都不匹配，则它的值字段被设置为 noSuchInstance。
- 否则，值字段被设置为变量值。

如果由于其他原因导致变量名处理过程的失败，则无法返回变量值。这时，应答实体将返回一个 error-status 字段值为 genErr，并在 error-index 字段中指出问题的变量的应答 PDU。

如果生成的应答 PDU 中的消息尺寸过大，超过了指定的最大限度，则生成的 PDU 被丢弃，并用一个 error-status 字段值为 tooBig，error-index 字段值为 0，变量绑定字段为空的新的 PDU 应答。

允许部分应答是对 GetRequest 的重要改进。在 SNMPv1 中，只要有一个变量值取不回来，所有的变量值就都不能返回。在这种情况下，发出操作请求的管理者往往只能将命令拆分为多条只取单个变量值的命令。相比之下，SNMPv2 的操作效率得到了很大提高。

（2）GetNextRequest PDU

SNMPv2 的 GetNextRequest PDU 的语法和语义都与 SNMPv1 的 GetNextRequest PDU 相同。与 GetRequestPDU 相同，两个版本的差别是对应答的处理。SNMPv1 的 GetNextRequest

是原子操作：要么所有的值都返回，要么一个也不返回，而 SNMPv2 能够部分地对 GetNextRequest 操作进行应答。

在 SNMPv2 中，按照以下规则处理 GetNextRequest 变量绑定字段中的每个变量来构造应答 PDU：

- 确定被指名的变量下一个变量，将该变量名和它的值成对地放入结果变量绑定字段中。
- 如果被指定的变量之后不存在变量，则将被指定的变量名和错误代码 endOfMibView 成对地放入结果变量绑定字段中。

如果由于其他原因导致变量名处理过程的失败，或者是产生的结果太大，处理过程与 GetRequest 相同。

（3）GetBulkRequest

SNMPv2 的一个主要改进是 GetBulkRequest PDU。这个 PDU 的目的是尽量减少查询大量管理信息时所进行的协议交换次数。GetBulkRequest PDU 允许 SNMPv2 管理者请求得到在给定的条件下尽可能大的应答。

GetBulkRequest 操作利用与 GetNextRequest 相同的选择原则，即总是顺序选择下一个对象。不同的是，利用 GetBulkRequest，可以选择多个后继对象。

GetBulkRequest 操作的基本工作过程如下：GetBulkRequest 在变量绑定字段中放入一个（N+R）个变量名的清单。对于前 N 个变量名，查询方式与 GetNextRequest 相同。即对清单中的每个变量名，返回它的下一个变量名和它的值，如果没有后继变量，则返回原变量名和一个 endOfMibView 的值。

GetBulkRequest PDU 有两个其他 PDU 所没有的字段，non-repeaters 和 max-repetitions。

non-repeaters 字段指出只返回一个后继变量的变量数。max-repetitions 字段指出其他的变量应返回的最大的后继变量数。为了说明算法，我们定义：

L = 变量绑定字段中的变量名数量

N = 只返回一个后继变量的变量名数

R = 返回多个后继变量的变量名数

M = 最大返回的后继变量数

在上述变量之间存在以下关系：

N = MAX [MIN (non-reperters, L), 0]

M = MAX [max-repetitions, 0]

$R = L - N$

如果 N 大于 0，则前 N 个变量与 GetNextRequest 一样被应答。如果 R 大于 0 并且 M 大于 0，则对应后面的 R 个变量，返回 M 个后继变量。即，对于每个变量：

- 获得给定变量的后继变量的值。
- 获得下一个后继变量的值。
- 反复执行上一步，直至获得 M 个对象实例。

如果在上面的过程中的某一点，已经没有后继变量，则返回 endOfMibView 值，在变量名处，返回最后一个后继变量，如果没有后继变量，则返回请求中的变量名。

利用这个规则，能够产生的 name-value 对的数量是 $N+(M\times R)$。后面的 $(M\times R)$ 对在应答 PDU 中的顺序可描述为：

for i : = 1 to M do

for r : = 1 to R do

retrieve i-th successor of (N+r)-th variable

即，返回的后继变量是一行一行的，而不是先返回第一个变量的所有后继变量，再返回第二个变量的所有后继变量等。

GetBulkRequest 操作解除了 SNMP 的一个主要限制，即不能有效地检索大块数据。此外，利用这个功能可以减小管理应用程序的规模。管理应用程序自身不需要关心组装在一起的请求的细节。不需要执行一个试验过程来确认请求 PDU 中的 name-value 对的最佳数量。并且，即使 GetBulkRequest 发出的请求过大，代理者也会尽量多地返回数据不是简单地返回一个 tooBig 的错误消息。为了获得缺少的数据，管理者只需简单地重发请求，而不必将原来的请求改装为小的请求序列。

（4）SetRequest

SetRequest PDU 由管理者发出，用来请求改变一个或多个对象的值。接收实体用一个包含相同 request-id 的 Response PDU 应答。与 SNMPv1 相同，SetRequest 操作是原子操作，即或者更新所有被指名的变量，或者所有的都不更新。如果接收实体能够为被指名的所有变量设置新值，则 Response PDU 返回与 SetRequest 相同的变量绑定字段。只要有一个变量值没设置成功，就不更新任何值。

SetRequest 的变量绑定分两个阶段处理。在第一阶段，确认每个绑定对。如果所有的绑定对都被确认，则进入第二阶段——改变每个变量。即每个变量的 set 操作都在第二阶段进行。

在第一阶段中，对每个绑定对进行以下确认，直至所有的都成功或遇到一个失败。失败的原因有：不可访问（noAccess）、无法建立或修改（notWritable）、数据类型不一致（wrongType）、长度不一致（wrongLength）、ASN.1 编码不一致、变量值有问题（wrongValue）、变量不存在且无法建立（noCreation）等。如果任意一个变量遇到以上情况，则返回一个在 error-status 字段给出上述错误代码，在 error-index 字段给出有问题的变量的序号的应答 PDU。与 SNMPv1 相比，提供了更多的错误代码。为管理站更容易地确定失败的原因提供了方便。

如果在确认阶段没有遇到问题，则进入第二阶段——更新在变量绑定字段中被指名的所有的变量。不存在的变量需要建立，存在的变量被赋予新值。只要遇到任何失败，则所有的更新都被撤销，并且返回一个 error-status 字段值为 commitFailed 的应答 PDU。

（5）SNMPv2 Trap

SNMPv2 Trap PDU 由一个代理者实体在发现异常事件时产生并发给管理站。与 SNMPv1 相同，它用于向管理站提供一个异步的通报以便报告重要事件。但它的格式与 SNMPv1 不同，与 GetRequest、GetNextRequest、GetBulkRequest、SetRequest 和 InformRequest PDU 拥有相同的格式。变量绑定字段用于容纳与陷阱消息有关的信息。Trap PDU 是一个非认证消息，不要求接收实体应答。

（6）InformRequest

InformRequest PDU 由一个管理者角色的 SNMPv2 实体应它的应用的要求发给另一个管理者角色的 SNMPv2 实体，请求后者向某个应用提供管理信息。与 SNMPv2 Trap PDU 类似，变量绑定字段被用于传送相关的信息。

收到 InformRequest 的实体首先检查承载应答 PDU 的消息尺寸，如果消息尺寸超过限度，用一个含有 tooBig 错误代码的 Response PDU 应答。否则，接收实体将 PDU 中的内容转到信息的目的地（某个应用），同时对发出 InformRequest 的管理者用 error-status 字段值为 noError 的 Response PDU 进行应答。

第 15 章　IPv6

我们在第 3 章讲述的 IP 协议的版本是第 4 版。IP 第 4 版作为网络的基础设施而广泛地应用在 Internet 和难以计数的小型专用网络上。尽管 IPv4 是一个非常成功的协议，它可以把数十个或数百个网络上的数以百计或数以千计的主机连接在一起，但它现在越来越显得不适应 Internet 的发展了。本章将讲述 IPv4 存在哪些问题，并介绍 IP 协议的新版本，版本 6。

15.1　IPv4 的不足与缺点

IPv4 的确是一个非常健壮的协议，并已经证明了它能够连接小至几个节点，大至 Internet 上难以计数的主机。目前，几乎所有的网络都在使用 IP 协议进行通信，而正是因为 IP 协议应用的如此广泛，它的改变将对所有使用 IP 的人产生重大影响。早在 20 世纪 80 年代初期人们就意识到了升级的需求，因为当时已经发现 IP 地址空间随着 Internet 的发展只能支持很短的时间。本节将介绍必须升级 IP 的原因以及可以同时改进之处，其中包括：

（1）地址空间的局限性：IP 地址空间的危机由来已久，并正是升级的主要动力。

（2）性能：尽管 IP 表现得不错，一些源自 20 年甚至更早以前的设计还能够进一步　改进。

（3）安全性：安全性一直被认为是由网络层以上的层负责，但它现在已经成为 IP 的下一个版本可以发挥作用的地方。

（4）配置：对于 IPv4 节点的配置一直比较复杂。现在笔记本等 IP 主机移动性的增强要求当主机在不同网络间移动和使用不同的网络接入点时能提供更好的配置支持。

下面具体说明 IPv4 的这些问题。

15.1.1　IP 地址空间危机

Internet 经历了飞速的发展，在过去的十多年间，连接到 Internet 的网络数量每隔不到一年的时间就会增加一倍。但即便是这样的发展速度，也并不足以导致 20 世纪 90 年代后期 IP 地址的匮乏。IP 地址为 32 位长，地址空间可能具有多于 40 亿的地址。导致 IP 地址耗尽的主要原因不在于 IP 地址的长度，而在于 IP 地址的结构。IP 地址采用分级地址结构，一个 IP 地址由网络地址部分和主机地址部分构成。通过使用分级地址格式，即每台主机首先依据它所连接的网络进行标识。IP 可支持简单的路由协议，主机只需要了解彼此的 I P 地址，就可以将数据从一台主机上传输到另一台主机。这种分级地址把地址分配的工作交给了网络管理员。到网络外的数据依据网络地址进行路由，在数据到达目的主机所连接的路由器之前无需了解主机地址。



如果不使用这种分级地址结构，而通过一个中央授权机构顺序化地为每台主机指派地址可能会使地址指派更加高效，但是这几乎使所有其他的网络功能不可行。例如，路由将实质上不可行。

IP 地址被分为 A、B、C、D、E 五类，只有前三类用于 IP 网络。近年来，随着 Internet 的膨胀，地址资源已显得越来越紧张。尽管人们已采用子网掩码和无域间路由等技术减少了 B 类地址的消耗，减轻了 IP 地址的浪费，但并没有从根本上解决问题。

15.1.2　IP 性能问题

IP 刚开始时，其主要目的在于为在异种网络间进行数据的可靠、健壮和高效传输提供最佳机制，从而实现不同计算机的互操作。在很大程度上 IP 实现了此目标，但这并不意味着 IP 不能加以改进。近几年，随着 Internet 及其应用的发展，对改进 IP 的要求越来越紧迫。在升级中主要考虑最大传输单元、最大包长度、IP 头的设计、校验和的使用、IP 选项的应用等议题。针对这些议题已经提出了专门建议并已引入 IPv6 中，这将有利于提高 IPv6 的性能并改进 IPv6 作为继续高速发展的网络基础的能力。

15.1.3　IP 安全性问题

刚开始时连入 Internet 的都是侧重于研究与开发的机构，当时安全性不是一个主要问题。更重要的是，很久以来人们认为安全性问题在网络协议栈的低层并不重要，应用安全性的责任仍交给应用层。在这种情况下，IPv4 几乎没有采取任何的安全保护措施，只具备最少的安全性选项。这对于目前开放的 Internet 显然是不适应的。

15.1.4　配置问题

在 IP 协议的早期，大部分使用 IP 协议接入 Internet 的计算机都是大型的计算机，它们与 Internet 的连接基本上是静态的。且使用这些计算机的人员都是熟悉 IP 协议的计算机专业研究人员。但现在则完全不同，人们可以选择任意的 ISP，他们可能携带笔记本电脑从一个网络转移到另一个网络，而且现在的无线网络技术可能使得这种网络间的移动是不易察觉的。而且现在使用计算机和网络的人员可能根本不懂计算机和网络。尽管动态主机配置协议（DHCP）可以允许系统在启动时通过服务器获取其正确和完整的 IP 网络配置，但主机（无论是移动的还是固定的）仍然依赖于到网络的单点连接。

15.1.5　IP 协议的升级策略

IPv4 的问题还不止上面列举的几个，例如还有：网络越多意味着路由表越大，同时导致路由器的性能下降；难以实现的 IPv4 选项意味着这些选项中实现的功能对用户不可用。如果只是简单地倍增 IP 地址的长度而不修改协议的其他部分，那么所有的 TCP/IP 协议栈将需要同时更新。尽管这种改变已经相对简单，但是由于错误配置而导致的系统瘫痪仍将产生巨大影响。对于拥有许多用户和主机的大型机构，这绝不是一件简单的事情。更复杂的是，有些系统中有许多是比较老的或过时的甚至是已经废弃的系统，在这些系统上运行的网络软件可

能已经过期并且没有人再提供支持。任何对于现有系统进行升级的请求都可能导致混乱。对于 IPv4 的修补，无论是临时加入一个补丁还是用另一个重新设计的协议来替换，都将导致混乱。与其他方法相比，升级不会带来更多的痛苦。因此 IPv6 协议规范在 1995 年底提交 IETF 并获得批准。各软件厂商也逐渐开始提供对 IPv6 的支持，各种 IPv6 的试验骨干网也已建立。

15.2　改进 IPv4 的各种努力

自从意识到 IPv4 的不足，各种改进 IPv4 的努力就在不断进行。本节将列出改进 IPv4 应该考虑的问题以及现有的各种努力。

15.2.1　Internet 发展的问题

对 IPv4 的改进应当集中在如下几个方面：路由与寻址、多协议体系结构、安全性体系结构、流量控制等。下面就每个问题展开讨论。

15.2.1.1　寻址与路由

地址空间毫无疑问已经是一个问题，而路由表的膨胀也值得密切注意。不仅 IPv4 的地址空间将耗尽，而且在此之前可能 IPv4 的路由算法已经无法适应如此大数量的网络。

对于寻址方案可能的修改包括使用现有的 32 位地址作为一个非全球惟一的标识。即，在网络中不互通的部分间地址可以重用。例如，把全球分为几个不同的域，这使得一个主机地址在每个域中都可以使用一次，而域间的互操作将通过协议网关在数据进行域间切换时重写其地址而进行。另一种寻址方案是只增加主机地址字段的长度，并集成一个管理域作为网络地址的一部分。第三种方案是使用让路由器将主机地址与管理域映射的连接策略扩展主机地址字段，并将整个字段作为一个非层次地址空间。

15.2.1.2　多协议体系结构

对互操作的 OSI 传输与 TCP/IP 业务流的支持是需要进一步开发的一个重要方面。Internet 的连接意味着一个主机必须具备一个 IP 地址。如果没有一个 IP 地址并且没有运行 IP，那么将不能上网。但 TCP/IP 应该而且可以包含或借鉴其他协议。而互操作性，尤其是应用之间而不是低层之间的互操作性是有益的。

15.2.1.3　安全性

美国国防部对于重点研究和开发工作的投资导致了 IP 的产生。但是，商用 Internet 在安全性需求上与军队的网络是不同的，并且向一个协议集中添加安全性要比从头建设一个安全性协议难得多。安全服务的一个建议是根据不同的用户名进行身份验证并加以访问控制。同时还提出了关于一致性的强制措施，其中包含了一些方法来防止传输过程中数据被修改以及对于传输源的欺骗和抵御重播攻击。其他的服务包括机密性、不可再现性（使用数字签名来防止发送方拒绝承认发送了某段数据）和通过拒绝对于某些服务的攻击以实现保护。

15.2.1.4 流量控制

IPv4 是一个无连接协议，但一些进程（例如语音和图像）需要一定程度的流量控制以正常工作。在 IPv4 中定义了服务类型 TOS 字段，但该字段不仅没有得到广泛应用而且现在连如何实现都不清楚。

15.2.2 各种努力

IPv6 并不是 IPv4 的惟一的改进升级方案。

IETF 在 1994 年考虑了三个主要提案。RFC1347，含有更多地址的 TCP 和 UDP（TUBA）是其中之一。TUBA 是一个简单的 Internet 寻址和路由协议，可以认为是简单地用 OSI 网络互联协议和无连接网络协议（CLNP）替换了 IP。CLNP 中使用了网络服务访问点（NSAP）地址，该地址可以是任意长度，从而提供了足够的地址空间。另一个提案在 1992 年以 IPv7 出现，并在 1993 年的 RFC1475 中详细描述，其标题为“TP/IX：下一代的 Internet”。TP/IX 使用 64 位地址，并在分级结构中加入了位于各单位之上的寻址层以用于管理。IPv7 的 8 字节地址中有 3 个字节用于管理域，3 个字节用于各单位的网络，另 2 个字节用于标识主机。IPv7 包头在对 IPv4 的包头进行简化的同时，也加入了转发路由标识符，使得中介路由器可以根据它来决定如何处理数据包。第三种提案有时称为 IP 中的 IP，或 IP 封装。在这个提案中，IP 包含两层：一层用于全球骨干网络，而另一层用于比较有限的范围。在有限范围内仍然使用 IPv4，但骨干网络中使用不同地址的新的一层。后来这种提案不断演变并与其他协议相融合从而产生了简单增强 IP（SIPP）。SIPP 经过一些修改之后，被 IESG 接受作为 IPng 的基础。

1995 年发布的 RFC1752 指出了 IPng 将来的样子。该协议提案中包括一个拥有分级地址结构的简化的头结构、包一级的身份验证和加密功能以及即插即用的自动配置功能。IPng 基于 SIPP 的 128 位地址。该 RFC 中的其他部分为 Internet 研究小组解决 IPv4 中的问题提供了非常好的历史资料，同时也提供了对于三个竞争者：TUBA、CATNIP 和 SIPP 的详细的分析。

与 IPv6 协议相关的 RFC 是在 1996 年和 1998 年发表的，在后面的各节将详细介绍 IPv6 的机制。

15.3 IPv6 对 IPv4 的改进

IPv6 的变化体现在以下五个重要方面：扩展地址、简化头格式、增强对于扩展和选项的支持、流标记和身份验证和保密。IPv6 的这些改进都解决了上节中提到的改进 IPv4 应该考虑和解决的各种问题。IPv6 的扩展地址使得 Internet 可以继续增长而无需考虑地址资源的消耗，该地址结构对于提高路由效率有所帮助；包头的简化减少了路由器上所需的处理过程，从而提高了路由的效率；同时，改进对头扩展和选项的支持意味着可以在几乎不影响普通数据包和特殊包路由的前提下适应更多的特殊需求；流标记办法为更加高效地处理包流提供了一种机制，这种办法对于实时应用尤其有用；身份验证和保密的改进使得 IPv6 更加适用于那些要求对敏感信息和资源特别对待的应用。下面来看 IPv6 在各方面采用的机制。

15.3.1　扩展地址

IPv6 的地址结构中除了把 32 位地址空间扩展到了 128 位外，还对 IP 主机可能获得的不同类型地址作了一些调整。IPv6 中取消了广播地址而代之以任意点播地址。IPv4 中用于指定一个网络接口的单播地址和用于指定由一个或多个主机侦听的多播地址基本不变。

15.3.2　简化的包头

IPv6 的包头有 8 个字段。它与 IPv4 包头的不同在于，IPv4 中包含至少 12 个不同字段，且长度在没有选项时为 20 字节，但在包含选项时可达 60 字节。IPv6 使用了固定格式的包头并减少了需要检查和处理的字段的数量，这将使得路由的效率更高。包头的简化使得 IP 的某些工作方式发生了变化。一方面，所有包头长度统一，因此不再需要包头长度字段。此外，通过修改包分段的规则可以在包头中去掉一些字段。最后，去掉 IP 头校验和不会影响可靠性，这主要是因为头校验和将由更高层协议（UDP 和 TCP）负责。

15.3.3　对扩展和选项支持的改进

在 IPv4 中可以在 IP 头的尾部加入选项。与 IPv4 不同的是，IPv6 把选项加在单独的扩展头中。通过这种方法，选项头只有在必要的时候才需要检查和处理。

15.3.4　流标记

在 IPv4 中，对所有包大致同等对待，路由器按照自己的方式对每个包单独进行处理，不会考虑不同的包之间的关系。IPv6 实现了流的概念。流指的是从一个特定源发向一个特定目的地的包序列，源点希望中间路由器对这些包进行特殊处理。因此，路由器需要有办法区分一个包是否属于同一个流。这样路由器就可以对流中的包进行高效处理。

15.3.5　身份验证和保密

IPv4 几乎没有对数据传输的安全需求作任何考虑，而是将该任务交给了上层协议或应用程序。IPv6 定义一套完整的安全保障措施。IPv6 使用两种安全性扩展：IP 身份验证头（AH）和 IP 封装安全数据（ESP）。数据包摘要功能通过对包的安全可靠性的检查和计算来提供身份验证功能。发送方计算数据包摘要并把结果插入到身份验证头中，接收方根据收到的数据包摘要重新进行计算，并把计算结果与 AH 头中的数值进行比较。如果两个数值相等，接收方可以确认数据在传输过程中没有改变；如果不相等，接受方可以推测出数据或者是在传输过程中遭到了破坏，或者是被某些人进行了故意的修改。封装安全性机制可以用来加密 IP 包的数据，或者在加密整个 IP 包后以隧道方式在 Internet 上传输。其中的区别在于，如果只对包的数据进行加密的话，包中的其他部分（包头）将公开传输。这意味着破译者可以由此确定发送主机和接收主机以及其他与该包相关的信息。使用 ESP 对 IP 进行隧道传输意味着对整个 IP 包进行加密，并由作为安全性网关操作的系统将其封装在另一 IP 包中。通过这种方法，加密的 IP 包中的所有细节均被隐藏起来。这种技术是创建虚拟专用网（VPN）的基础，

它允许各机构使用 Internet 作为其专用骨干网络来共享敏感信息。

15.4　IPv6 数据包结构

本节将给出 IPv6 协议数据包的具体格式，并对其中的部分新的概念和机制作出详细说明。

15.4.1　IPv6 数据包的结构

在 IPv4 中，所有包头以 32 位为单位，即 4 个八位字。而在 IPv6 中，包头以 64 位为单位，且包头的总长度是 40 个八位字。图 15.1 是 IPv6 数据包的格式示意图。

版本字段的长度是 4 位，这是为了与 IPv4 兼容。在同一个网络上可以同时存在两种不同版本的 IP 协议，通过该字段区分数据包是哪个版本的，并进行不同的处理。业务流类别字段的长度为 8 位，表示数据包所属的类别，路由器可以为不同类别的数据包提供不同的服务。默认值为 0。流标签字段的长度为 20 位，用于标识属于同一业务流的包。一个节点可以同时作为多个业务流的发送源。流标签和源节点地址惟一标识了一个业务流。数据长度字段的长度为 16 位，其中包括包数据的长度，按 8 位字计。也即是 IPv6 头后的包中包含的字节数。这意味着在计算数据长度时包含了 IPv6 扩展头的长度。下一个头字段指出了 IPv6 头后所跟的头字段中的协议类型。其作用类似于 IPv4 的协议字段，可以用来指出高层是 TCP 还是 UDP，但它也可以用来指明 IPv6 扩展头的存在。跳极限字段的长度为 8 位。每当一个路由器对包进行一次转发之后，这个字段就会减 1。如果该字段达到 0，这个包就将被丢弃。这和 IPv4 中的 TTL 字段类似。但和 IPv4 不同的是，IPv6 中的极限跳数字段并不代表时间含义，那么对数据包超时的判断可以由高层协议完成。源 IP 地址和目的 IP 地址字段分别长 128 位，是两个 IPv6 的 IP 地址。

0	4	12	16	24　　31
版本	业务流类别	流标签		
净荷长			下一个头	跳极限
源IP地址				
目的IP地址				
数据区				

图 15.1　IPv6 数据包结构

15.4.2　IPv6 的服务类型和流标签

IPv4 中定义了一个服务类型字段，但该字段在实现中几乎都被忽略了，无法起到为不同的数据包提供不同服务的作用。IPv6 中定义了一个业务流类型字段。使用业务流类别的目的在于允许发送业务流的源节点和转发业务流的路由器在包上加上标记，并进行除默认处理方法之外的不同处理。一般来说，在所选择的链路上，可以根据开销、带宽、延时或其他特性而对包进行特殊的处理。

使用 IPv4 的路由器在转发 IP 数据包时，是根据数据包本身的目的地址选择路径进行转发的。即使后面的数据包和前面的是到达同一个地址的，有可能它们到达目的地所经过的路径会不相同。这对于适应网络突发事件来说是个好办法，因为突发事件意味着任何一条路由都可能在任何时间出现故障，但只要两主机间存在某些路由则可以进行数据的交互。但是，这种方法的效率可能不太高，因为数据包并不是孤立的，一个主机与另一个主机通信时发送的数据包绝大多数时都不止一个。如果已经为这次通信的第一个数据包选好了路径，那么该通信的以后的所有包都可以通过这条路径到达目的地，而不用为每个数据包都进行一次选择。IPv6 中流的概念将有助于解决类似问题。IPv6 头字段中的流标签把单个包作为一系列源地址和目的地址相同的包流的一部分。同一个流中的所有包具有相同的流标签。

15.4.3　IP 数据包的分片

IPv6 的分片只能由源节点和目的节点进行，这样就简化了包头并减少了用于路由的开销。Ipv4 的逐跳分片是一种有害的方法。首先，它在端到端的分片中将产生更多的分片。此外在传输中，一个分片的丢失将导致所有分片重传。虽然 IPv4 在使用了分片之后，不论中间的网络是什么类型，不同类型网络上的节点都可以互操作，源节点无需了解任何有关目的节点网络的信息，同时也无需了解它们之间的网络信息。这一直被认为是一个不错的特性，由于不需要节点或路由器存储信息或记录整个 Internet 的结构，从而可以获得很好的扩展性。但另一方面，它也为路由器带来了性能方面的问题，对 IP 包进行分片消耗了沿途路由器和目的地的处理能力和时间。处理 IP 数据包标识、计算分片偏移值、真正把数据分片以及在目的地进行重装都会带来额外的开销。问题在于对于任何一个指定的路由器，虽然源节点能够了解链路的 MTU 是多大，但却没有办法事先知道整个路径的 MTU。然而，目前有两种方法可以减少或消除对于分片的需求。第一种方法可用在 IPv4 中，它使用一种叫做“路径 MTU 发现”的方法。通过这种方法，路由器可以向目的地发送一个包来报告该路由器上链路的 MTU 值。如果包到达了一条必须对其进行分片的链路，负责分片的路由器将使用 ICMP 回送一个数据包来指出分片路由器上链路的 MTU 值。这种过程可以重复进行直到路由器确定路径 MTU 为止。另一种方法是要求所有支持 IP 的链路必须能够处理一些合理的最小长度的包。换句话说，如果一个链路的 MTU 超过 20 字节，那么所有的节点都必须准备产生可观数量的分片包。另一方面，如果能够提出所有网络链路都可以适应的某个合理的长度，并把它设置为允许包长度的绝对最小值，那么就可以消灭分片。IPv6 中实际上同时使用了上面两种方法。



15.4.4 扩展头

IPv4 选项的问题在于改变了 IP 头的大小，它们需要特别的处理。路由器必须优化其性能，这意味着将为最普遍的包进行最佳性能的优化。这使得 IPv4 选项引发一个路由器把包含该选项的包搁置一边，等到有时间的时候再进行处理。IPv6 中实现的扩展头可以消除或至少大量减少选项带来的对性能的冲击。通过把选项从 IP 头中搬到数据中，路由器可以像转发无选项包一样来转发包含选项的包。除了规定必须由每个转发路由器进行处理的逐跳选项之外，IPv6 包中的选项对于中间路由器而言是不可见的。可用的选项除了减少 IPv6 包转发时选项的影响外，IPv6 规范使得对于新的扩展和选项的定义变得更加简单。在需要的时候可能还会定义其他的选项和扩展。

本节仅简单列出已定义的扩展：

（1）逐跳选项头。此扩展头必须紧随在 IPv6 头之后。它包含包所经路径上的每个节点都必须检查的选项数据。由于它需要每个中间路由器进行处理，逐跳选项只有在绝对必要的时候才会出现。到目前为止，已经定义了两个选项：巨型数据选项和路由器提示选项。巨型数据选项指明包的数据长度超过 IPv6 的 16 位数据长度字段。只要包的数据超过 65535 字节（其中包括逐跳选项头），就必须包含该选项。如果节点不能转发该包，则必须回送一个 ICMPv6 出错数据包。路由器提示选项用来通知路由器，IPv6 数据包中的信息希望能够得到中间路由器的查看和处理，即使这个包是发给其他某个节点的。

（2）路由头。此扩展头指明包在到达目的地途中将经过哪些节点。它包含包沿途经过的各节点的地址列表。IPv6 头的最初目的地址是路由头的一系列地址中的第一个地址，而不是包的最终目的地址。此地址对应的节点接收到该包之后，对 IPv6 头和路由头进行处理，并把包发送到路由头列表中的第二个地址。如此继续，直到包到达其最终目的地。

（3）分段头。此扩展头包含一个分段偏移值、一个“更多段”标志和一个标识符字段。用于源节点对长度超出源端和目的端路径 MTU 的包进行分段。

（4）目的地选项头。此扩展头代替了 IPv4 选项字段。目前，惟一定义的目的地选项是在需要时把选项填充为 64 位的整数倍。此扩展头可以用来携带由目的地节点检查的信息。

（5）身份验证头（AH）。此扩展头提供了一种机制，对 IPv6 头、扩展头和数据的某些部分进行加密的校验和的计算。

（6）封装安全性数据（ESP）头。这是最后一个扩展头，不进行加密。它指明剩余的数据已经加密，并为已获得授权的目的节点提供足够的解密信息。

15.5 IPv6 的寻址方式

IPv6 对 IP 协议的寻址方式做了彻底的修改，而不是仅仅扩大 IP 地址空间而已。IPv6 的这种改进不仅提高了地址分配的效率，同时还有助于提高 IP 路由的效率。本节将仔细研究 IPv6 的寻址方式。

15.5.1　地址结构与寻址模式

IPv4 与 IPv6 地址之间最明显的差别在于长度：IPv4 地址长度为 32 位，而 IPv6 地址长度为 128 位。IPv4 地址可以分为 2 至 3 个不同部分（网络标识符、节点标识符，有时还有子网标识符），IPv6 地址中拥有更大的地址空间，可以支持更多的字段。IPv6 地址有三类：单播、多播和泛播地址。单播和多播地址与 IPv4 的地址非常类似；但 IPv6 中不再支持 IPv4 中的广播地址，而增加了一个泛播地址。

IPv4 地址一般以 4 部分间点分的方法来表示，即 4 数字用点分隔。IPv6 地址的基本表达方式是 X:X:X:X:X:X:X:X，其中 X 是一个 4 位十六进制整数（16 位）。例如，下面是一些合法的 IPv6 地址：

1A8D:127D:35FC:ABCD:47BC:1743:0210:45DA

3910:0:0:0:0:0:0:1

请注意这些整数是十六进制整数。这是一种比较标准的 IPv6 地址表达方式，但这种方式由于数据太多不便于人阅读与记忆。还有另外两种更加清楚和易于使用的方式。某些 IPv6 地址中可能包含一长串的 0（就像上面的第二个例子一样）。当出现这种情况时，标准中允许用“空隙”来表示这一长串的 0。换句话说，地址

3910:0:0:0:0:0:0:1

可以表示为：

3910::1

这两个冒号表示该地址可以扩展到一个完整的 128 位地址。在这种方法中，只有当 16 位组全部为 0 时才可以用两个冒号取代，且两个冒号在地址中只能出现一次。在 IPv4 和 IPv6 的混合环境中可能有第三种方法。IPv6 地址中的最低 32 位可以用于表示 IPv4 地址，该地址可以按照一种混合方式表达，即 X:X:X:X:X:X:d.d.d.d，其中 X 表示一个 16 位整数，而 d 表示一个 8 位十进制整数。例如，地址

0:0:0:0:0:0:10.0.0.1

就是一个合法的 IPv4 地址。把两种可能的表达方式组合在一起，该地址也可以表示为：

::10.0.0.1

由于 IPv6 地址被分成两个部分——子网前缀和接口标识符，因此人们期待一个 IP 节点地址可以按照类似 CIDR 地址的方式表示为一个携带额外数值的地址，其中指出了地址中有多少位是掩码。即 IPv6 节点地址中指出了前缀长度，该长度与 IPv6 地址间以斜杠区分，例如：

1030:0:0:0:C9B4:FF12:48AA:1A2B/60

这个地址中用于路由的前缀长度为 60 位。

IPv6 寻址模型与 IPv4 很相似。每个单播地址标识一个单独的网络接口。IP 地址被指定给网络接口而不是节点，因此一个拥有多个网络接口的节点可以具备多个 IPv6 地址，其中任何一个 IPv6 地址都可以代表该节点。尽管一个网络接口能与多个单播地址相关联，但一个单播地址只能与一个网络接口相关联。每个网络接口必须至少具备一个单播地址。但也有例外。

在 IPv4 中，所有的网络接口，其中包括连接一个节点与路由器的点到点链路（用许多拨



号 Internet 连接中），都需要一个专用的 IP 地址。随着许多机构开始使用点到点链路来连接其分支机构，每条链路均需要其自己的子网，这样一来消耗了许多地址空间。在 IPv6 中，如果点到点链路的任何一个端点都不需要从非邻居节点接受和发送数据的话，它们就可以不需要特殊的地址。即，如果两个节点主要是传递业务流，则它们并不需要具备 IPv6 地址。为每个网络接口分配一个全球惟一的单播地址的要求阻碍了 IPv4 地址的扩展。一个提供通用服务的服务器在高需求量的情况下可能会崩溃。因此，IPv6 地址模型中又提出：如果硬件有能力在多个网络接口上正确地共享其网络负载的话，那么多个网络接口可以共享一个 IPv6 地址。这使得从服务器扩展至负载分担的服务器群成为可能，而不再需要在服务器的需求量上升时必须进行硬件升级。

15.5.2　地址类型

IPv6 的地址可分为三种类型：

单播：一个单接口的标识符。送往一个单播地址的包将被传送至该地址标识的接口上。

泛播：一组接口（一般属于不同节点）的标识符。送往一个泛播地址的包将被传送至该地址标识的接口之一（根据路由协议对于距离的计算方法选择“最近”的一个）。

多播：一组接口（一般属于不同节点）的标识符。送往一个多播地址的包将被传送至有该地址标识的所有接口上。

IPv6 将 IPv4 中的广播地址类型去除了，并添加一种泛播地址类型。

15.5.2.1　IPv4 广播地址的问题

广播用来携带去向多个节点的信息或被哪些不知信息来自何方的节点用来发出请求。但是，广播可能会给网络性能设置障碍。同一网络链路上的大量广播意味着该链路上的每个节点都必须处理所有广播，其中绝大部分节点最终都将忽略该广播，因为该信息与自己无关。把广播在子网之间进行转发将导致更多的问题，因为路由器上将充斥着这种业务流。

IPv6 对此的解决办法是使用一个“所有节点”多播地址来替代那些必须使用广播的情况，同时，对那些原来使用了广播地址的场合，则使用一些更加有限的多播地址。通过这种方法，对于原来由广播携带的业务流感兴趣的节点可以加入一个多播地址，而其他对该信息不感兴趣的节点则可以忽略发往该地址的包。广播从来不能解决信息穿越 Internet 的问题，如路由信息，而多播则提供了一个更加可行的方法。

15.5.2.2　单播地址

单播地址标识了一个单独的 IPv6 接口。一个节点可以具有多个 IPv6 网络接口。每个接口必须具有一个与之相关的单播地址。单播地址包含了一段信息，这段信息包含在 128 位字段中：该地址可以完整地定义一个特定的接口。此外，地址中数据可以解释为多个小段的信息。

IPv6 地址本身可以为节点提供关于其结构的或多或少的信息，这主要根据是由谁来观察这个地址以及观察什么。节点可能只需简单地了解整个 128 位地址是一个全球惟一的标识符，而无须了解节点在网络中是否存在。另一方面，路由器可以通过该地址来决定，地址中的一部分标识了一个特定网络或子网上的一个惟一节点。

最简单的方法是把 IPv6 地址作为不加区分的一块 128 位的数据，而从格式化的观点来看，可把它分为两段，即接口标识符和子网前缀。

IPv6 单播地址包括下面几种类型：可集聚全球地址、未指定地址或全 0 地址、回返地址、嵌有 IPv4 地址的 IPv6 地址、基于供应商和基于地理位置的供应商地址、OSI 网络服务访问点（NSAP）地址和网络互联包交换（IPX）地址。

15.5.2.3　多播地址

IPv6 多播地址的格式不同于 IPv6 单播地址，它将 128 位的地址分为几个固定长度的字段。地址格式中的第 1 个字节为全 1，表示其为多播地址。多播地址占了 IPv6 地址空间的整整 1/256。多播地址格式中除第 1 字节外的其余部分，包括如下三个字段：

（1）标志字段：长 4 位。目前只指定了第 4 位，该位用来表示该地址是由 Internet 编号机构指定的熟知的多播地址，还是特定场合使用的临时多播地址。如果该标志位为 0，表示该地址为熟知地址；如果该位为 1，表示该地址为临时地址。其他 3 个标志位保留将来用。

（2）范围字段：长 4 位，用来表示多播的范围。即，多播组是只包括同一本地网、同一站点、同一机构中的节点，还是包括 IPv6 全球地址空间中任何位置的节点。该 4 位的可能值为 0～15。

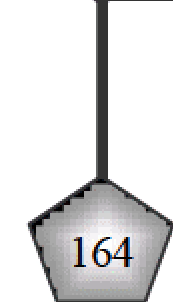
（3）组标识符字段：长 112 位，用于标识多播组。根据多播地址是临时的还是熟知的以及地址的范围，同一个多播标识符可以表示不同的组。永久多播地址用指定的赋予特殊含义的组标识符，组中的成员既依赖于组标识符，又依赖于范围。

15.5.2.4　泛播地址

多播地址在某种意义上可以由多个节点共享。多播地址成员的所有节点均期待着接收发给该地址的所有包。一个连接 5 个不同的本地以太网网络的路由器，要向每个网络转发一个多播包的副本（假设每个网络上至少有一个预订了该多播地址）。泛播地址与多播地址类似，同样是多个节点共享一个泛播地址，不同的是，只有一个节点期待接收给泛播地址的数据包。泛播对提供某些类型的服务特别有用，尤其是对于客户机和服务器之间不需要有特定关系的一些服务，例如域名服务器和时间服务器。

泛播地址被分配在正常的 IPv6 单播地址空间以外。因为泛播地址在形式上与单播地址无法区分开，一个泛播地址的每个成员，必须显式地加以配置，以便识别泛播地址。

了解如何为一个单播包确定路由，必须从指定单个单播地址的一组主机中提取最低的公共路由命名符。即，它们必定有某些公共的网络地址号，并且其前缀定义了所有泛播节点存在的地区。比如一个 ISP 可能要求它的每一个用户机构提供一个时间服务器，这些时间服务器共享单个泛播地址。在这种情况下，定义泛播地区的前缀，被分配给 ISP 作再分发用。发生在该地区中的路由是由共享泛播地址的主机的分发来定义的。在该地区中，一个泛播地址必定带有一个路由项：该路由项包括一些指针，指向共享该泛播地址的所有节点的网络接口。上述情况下，地区限定在有限范围内。泛播主机也可能分散在全球 Internet 上，如果是这种情况的话，那么泛播地址必须添加到遍及世界的所有路由表上。



15.6 Ipv6 的安全性

IPv6 通过使用身份验证头（AH）和封装安全性数据（ESP）头来实现身份验证和安全性，包括安全密码传输、加密和数据包的数字签名。本节将研究 IP 安全性体系结构以及在 IPv6 中的实现机制。

15.6.1 IP 协议的安全目标

IPv4 的目的只是作为简单的网络互通协议，因而其中没有包含任何安全特性。对于安全性，可以定义如下 3 个公认的目标。

（1）身份验证：能够可靠地确定接收到的数据与发送的数据一致，并且确保发送该数据的实体与其所宣称的身份一致。

（2）完整性：能够可靠地确定数据在从源到目的地传送的过程中没有被修改。

（3）机密性：确保数据只能为预期的接收者使用或读出，而不能为其他任何实体使用或读出。

完整性和身份验证经常密切相关，而机密性有时使用公共密钥加密来实现，这样也有助于对源端进行身份验证。IPv6 的 AH 和 ESP 头有助于在 IP 上实现上述目标。很简单，AH 为源节点提供了在包上进行数字签名的机制。AH 之后的数据都是纯文本格式，可能被攻击者截取。但是，在目的节点接收之后，可以使用 AH 中包含的数据来进行身份验证。另一方面，可以使用 ESP 头对数据内容进行加密。ESP 头之后的所有数据都进行了加密，ESP 头为接收者提供了足够的数据以对包的其余部分进行解密。

Internet 安全性的问题在于很难创建安全性，尤其是在开放的网络中，包可能经过任意数量的未知网络，任一个网络中都可能有包嗅探器在工作，而任何网络都无法察觉。在这样的开放环境中，即使使用了加密和数字签名，安全性也将受到严重的威胁。对 IP 业务流的攻击也包括诸如侦听之类，致使从一个实体发往另一个实体的数据被未经授权的第三个实体所窃取。密钥管理问题则更加复杂。为使身份验证和加密更可靠，IP 安全性体系结构要求使用密钥。如何安全地管理和分配密钥，同时又能正确地将密钥与实体结合以避免中间者的攻击，这是 Internet 业界所面临的最棘手的问题之一。

15.6.2 IPsec

IPsec 的目标是提供既可用于 IPv4 也可用于 IPv6 的安全性机制，该服务由 IP 层提供。一个系统可以使用 IPsec 来要求与其他系统的交互以安全的方式进行——通过使用特定的安全性算法和协议。IPsec 提供了必要的工具，用于一个系统与其他系统之间对彼此可接受的安全性进行协商。即系统可能有多个可接受的加密算法，这些算法允许该系统使用它所倾向的算法和其他系统协商，但如果其他系统不支持它的第一选择，则它也可以接受某些替代算法。IPsec 中可能提供如下安全性服务：

（1）访问控制。如果没有正确的密码就不能访问一个服务或系统。可以调用安全性协

议来控制密钥的安全交换，用户身份验证可以用于访问控制。

（2）无连接的完整性。使用 IPsec，有可能在不参照其他包的情况下，对任一单独的 IP 包进行完整性校验。此时每个包都是独立的，可以通过自身来确认。此功能可以通过使用安全散列技术来完成，它与使用检查数字类似，但可靠性更高，并且更不容易被未授权实体所篡改。

（3）数据源身份验证。IPsec 提供的又一项安全性服务是对 IP 包内包含的数据的来源进行标识。此功能通过使用数字签名算法来完成。

（4）对包重放攻击的防御。作为无连接协议，IP 很容易受到重放攻击的威胁。重放攻击是指攻击者发送一个目的主机已接收过的包，通过占用接收系统的资源，这种攻击使系统的可用性受到损害。为对付这种攻击，IPsec 提供了包计数器机制。

（5）加密。数据机密性是指只允许身份验证正确者访问数据，对其他任何人一律不准。它是通过使用加密来提供的。

（6）有限的业务流机密性。有时只使用加密数据不足以保护系统。只要知道一次加密交换的末端点、交互的频度或有关数据传送的其他信息，坚决的攻击者就有足够的信息来使系统混乱或毁灭系统。通过使用 IP 隧道方法，尤其是与安全网关共同使用，IPsec 提供了有限的业务流机密性。

通过正确使用 ESP 头和 AH 头，上述所有功能都有可能得以实现。

安全关联（SA）是 IPsec 的基本概念。安全性关联包含能够惟一标识一个安全性连接的数据组合。连接是单方向的，每个 SA 由目的地址和安全性参数索引（SPI）来定义。其中 SPI 说明使用 SA 的 IP 头类型，如 AH 或 ESP。SPI 为 32 位，用于对 SA 进行标识及区分同一个目的地址所链接的多个 SA。进行安全通信的两个系统有两个不同的 SA，每个目的地址对应一个。每个 SA 还包括与连接协商的安全性类型相关的多个信息。这意味着系统必须了解其 SA、与 SA 目的主机所协商的加密或身份验证算法的类型、密钥长度和密钥生存期。

IPsec 的数据传输可以有隧道模式和透明模式两种。两个通信的系统直接建立了 SA。其中一个系统产生数据，经过加密或者签名，然后发送给目的系统。而在接收方，首先对收到的数据包进行解密或者身份验证，把数据向上传送给接收系统的网络栈，由使用数据的应用进行最后的处理。两个主机之间的通信如同没有安全头一样简单，而且数据包实际的 IP 头必须要暴露出来以便进行路由，这种方式称为透明模式。而如果通信的两个系统之间不是直接进行通信的，而是发送方先把数据发送到一个安全网关，由它和另一个安全网关之间建立一条安全的通道并将数据发送到该网关，接收网关再将数据发送给目的系统。这个过程中，真正进行通信的两个系统的信息不会暴露在传输的 IP 头中。它们的通信好像是在一个安全的通道中进行的，所以称为隧道模式。

15.6.3　IPv6 安全头

IPsec 安全性服务完全通过 AH 和 ESP 头相结合的机制来提供。各安全头可以单独使用，也可以一起使用。如果一起使用多个扩展头，AH 应置于 ESP 头之前，这样，首先进行身份验证，然后再对 ESP 头数据解密。使用 IPsec 隧道时，这些扩展头也可以嵌套。AH 和 ESP 头既可以用于 IPv4，也可以用于 IPv6。本节将讨论这些安全性扩展头在 IPv6 中的使用方式

和工作机制。

15.6.3.1 AH 头

AH 的作用包括如下：

（1）为 IP 数据包提供强大的完整性服务，即 AH 可用于为 IP 数据包验证数据。

（2）为 IP 数据包提供身份验证，即 AH 可用于将实体与数据包内容相链接。

（3）如果在完整性服务中使用了公共密钥数字签名算法，AH 可以为 IP 数据包提供不可否认服务。

（4）通过使用顺序号字段来防止重放攻击。

AH 可以在隧道模式或透明模式下使用。它既可用于为两个节点间的简单直接的数据包传送提供身份验证和保护，也可用于对发给安全性网关或由安全性网关发出的整个数据包流进行封装。

IPv6 中的 AH 与其他扩展头一起使用时，必须置于那些将由中间路由器处理的扩展头之后，并在只能由数据包目的地处理的扩展头之前。在透明模式中，AH 保护初始 IP 数据包的数据，也保护在逐跳转发中不变化的部分 IP 头，如跳极限字段或路由扩展头。当 AH 用于隧道模式中时，使用方法与上不同，整个初始 IP 数据包以及传送中不变的封装 IP 头部分都应该保护。

15.6.3.2 ESP 头

ESP 头允许 IP 节点发送和接收数据经过加密的数据包。ESP 头可以提供下列几种服务：

（1）通过加密提供数据包的机密性。

（2）通过使用公共密钥加密对数据来源进行身份验证。

（3）通过由 AH 提供的序列号机制提供对抗重放服务。

（4）通过使用安全性网关来提供有限的业务流机密性。

ESP 头可以和 AH 结合使用。实际上，如果 ESP 头不使用身份验证的机制就应该将 AH 和 ESP 头一起使用。

ESP 头必须跟随在去往目的节点所途经的中间节点需要处理的扩展头之后，ESP 头之后的数据都可能被加密。ESP 既可用于隧道模式，也可用于透明模式。在透明模式中，如果有 AH，IP 头以及逐跳扩展头、路由扩展头或分段扩展头都在 AH 之前，其后跟随 ESP 头。任何目的地选项头可以在 ESP 头之前，也可以在 ESP 头之后，或者 ESP 头前后都有，而 ESP 头之后的扩展头将被加密。

在很多方面，仅仅是常规数据包带着加密数据从源端传送到目的端。某些情况下，适合在透明模式中使用 ESP。但是，这种模式使攻击者有可能研究两个节点之间的业务流，留意正在通信的节点、节点之间交换的数据量、交换的时间等。所有这些信息都可能为攻击者提供有助于对通信双方进行攻击的信息。类似前面描述的 AH 的情形，使用安全性网关是一种替代方法。安全性网关可以直接与节点连接，也可以链接到另一个安全性网关。单个节点可以在隧道模式中使用 ESP，即加密所有出境包，并封装到单独的 IP 数据包流中，再发送给安全性网关。然后网关解密业务流，并重新将原始 IP 数据包发往目的地。使用隧道模式时，ESP 头对整个 IP 数据包进行封装，并作为 IP 头的扩展将数据包定向到安全性网关。ESP 头与 AH 的结合也有几种不同方式，例如以隧道方法传送的数据包可能有透明模式的 AH。



15.7　IP 协议的升级对其他协议的影响

TCP/IP 协议族分为四层，现在对 IP 协议进行了升级，会不会对其他的各层协议造成影响，并要求其也进行相应的升级呢？本节将从整体上看这个问题，但不会就每种具体的协议进行讨论。

首先是物理网络层。物理层协议由于 IP 协议的升级而受到的影响很小。这是由于这些协议只是将上层数据包封装到物理层帧中。但这并不说明 IPv6 对物理层协议毫无影响。例如，ATM 使用类似点到点连接来跨越网络传送数据，对于需要将 IPv6 包交付多个节点的服务，ATM 需要格外注意。可能受 IPv6 影响的物理层问题还包括路径 MTU 发现及地址解析协议（ARP），这些协议需要修改以支持 128 位的 IPv6 地址。

其次是传输层。IP 层与传输层协议的关系最密切，因为传输层的实现是基于 IP 协议提供的服务的。传输层中，UDP 和 TCP 的伪头部都使用了源 IP 地址和目的 IP 地址，而且 TCP 的连接是由源节点和目的节点的 IP 地址和端口号来定义的。如果要与 IPv6 互操作，至少要修改 UDP 和 TCP，以适应 128 位 IPv6 地址。另外，因为 IPv6 支持移动 IP，而目前 TCP 在处理移动节点时有一点问题：确定 TCP 连接需要源节点和目的节点的 IP 地址。如果在 TCP 交互期间，一方或双方的 IP 地址有所改变，则连接的标识就会出现问题。移动节点从一个网络地址向另一个网络地址转换时就会出现这种情况，这种问题的产生是由于 TCP 至少在目前还没有机制能允许在连接中改变 IP 地址。如果一个节点收到的 TCP 段中的源 IP 地址与此 TCP 连接在建立时协商的地址不同，该节点将认为这个 TCP 段是属于另一个连接的。解决这个问题比简单地允许 TCP 连接支持网络地址转换要复杂许多。因为支持这样的地址转换将导致安全性漏洞：攻击者很容易冒充从一个网络向另一个网络转换的节点，如同授权的节点从一个网络向另一个网络转换一样。解决这样的问题将要求对 TCP 进行重大的升级，即需要引入机制使节点在其 IP 地址改变时能向其他节点证明自己。

最后是应用层。由于很多与 TCP/IP 协议相关的应用在其实现或参数设置都直接使用或允许直接使用 IP 地址。现在 IPv6 将地址增加到了 128 位，所以这些应用显然需要进行更新。而且，现在 IPv6 提供更好的的安全性、服务质量或其他特性的服务，有些应用希望使用 IPv6 服务，这样就需要更广泛的更新。

第 16 章　常见操作系统 TCP/IP 协议实现

本章将介绍目前常见的操作系统中的 TCP/IP 协议栈的实现机制。当然，无法概括各系统实现的所有细节问题，也无法包括所有的操作系统。只讨论 Windows 和 Unix/Linux 中 TCP/IP 协议栈的整体体系结构及一些重要的具体机制。由于它们的实现有很多相同之处，我们将大部分的篇幅都安排在讲述 Windows 的 TCP/IP 实现上，这主要是因为本书在第 2 部分讲述 Internet 编程技术时都是以 Windows 平台为例进行说明的。在这里先让读者了解 Windows 的 TCP/IP 的整体结构和功能，方便读者对本书第 2 部分的理解。对 Unix/Linux 中 TCP/IP 实现的详细信息，可以参考相关的文档和参考资料。

16.1　Windows 的 TCP/IP 实现

Windows 操作系统作为当今最主流的操作系统，对企业网络传输起着巨大的作用。Microsoft Windows 上的允许企业级互联和基于 Windows 平台之间的计算机建立连接。在 Windows 中添加 TCP/IP 能获得如下的功能：

（1）一个标准的、可以路由的企业网络互联协议，也是可用协议中最完整和最为大众所接受的。所有的现代网络操作系统都支持 TCP/IP，大型网络的大部分网络通信都依赖于 TCP/IP。

（2）一种连接不同系统的技术。许多标准的连接程序可以用来在不同系统之间访问和传输数据，包括文件传输协议（FTP）和 Telnet（一种终端仿真协议）。Windows 包括几个这样的标准程序。

（3）一个健壮的、可扩展的、跨平台的客户/服务器框架。Microsoft TCP/IP 提供 Windows 套接字接口，它是理想的客户机/服务器应用程序开发工具，能在其他厂商的与 Windows 套接字兼容的协议栈上运行。

（4）一种访问 Internet 的方法。Internet 包括成千上万的世界性网络、互联的研究机构、大学、图书馆和公司。

Windows TCP/IP 的核心协议元素、服务及它们之间的接口如图 16.1 所示。传输驱动程序接口（TDI）和网络设备接口规范（NDIS）是公共的，有关它们的描述可以从 Microsoft 公司得到。另外，还有许多高层接口可供用户模式的应用程序使用。最常用的有 Windows 套接字、远程过程调用（RPC）和 NetBIOS。欲知关于 Windows 套接字的更多信息，参见本书第 2 部分。

下面，按照自底向上的顺序逐一说明 Windows 平台下的 TCP/IP 协议栈。

NetBIOS 应用程序
RPC 应用程序
Win32 应用程序
Windows Socket 应用程序
应用和用户模式服务
应用接口
RPC
Wnet
WinInet
NetBIOS 支持
Windows 套接字
用户模式
核心模式
命名管道
重定向
服务器
NetBT
AFD
TDI
包分类器
TCP/UDP
IP
ICMP
IP转发器
IP转发器
IGMP
ARP
通信流控制
包调度器
包队列
包队列
包队列
包队列
包队列
NDIS
NDIS包装器
NDIS微端口包装
PPTP
Asynch
X.25
ISDN
X.25
帧中继
ATM
以太网
FDDI
令牌环

图 16.1　Windows TCP/IP 协议栈组成元素及服务接口

16.1.1　物理链路层

Windows 网络协议使用网络设备接口规范（NDIS）网卡驱动程序进行通信。开放系统互联（OSI）模型中数据链路层的大部分功能都在该协议栈中实现，这使得开发网卡驱动程序更简单。NDIS5.0 包括以下扩展功能：

- NDIS 电源管理（网络电源管理和网络唤醒需要此功能）。
- 即插即用。
- 对诸如 TCP 和 UDP 校验和之类任务的任务分载机制和快速包转发。
- 支持 QoS。
- 支持中间驱动器（广播式 PC），虚拟局域网（VLAN），面向 QoS 的包调度，NDIS（对 IEEE1394 网络设备的支持都需要此功能）。

当系统请求电源级别改变时，NDIS 就能切断网络适配器的电源。用户或系统都能启动该请求。例如：用户可能想使计算机进入睡眠状态，或者系统可能因为键盘或鼠标不活动而请求改变电源级别。另外，如果网络适配器支持的话，断开网络连线也能启动该请求。在这

种情况下，系统会在切断网络适配器电源之前等待一段可配置的时间，因为连接断开可能只是网络中临时线路改变的结果，并非真的断开电缆与网络的连接。

NDIS 电源管理策略的前提是没有网络活动。这意味着在切断网络适配器电源之前，所有上层网络组件必须同意该请求。如果网络上还存在活动会话或者打开的文件，断电请求就会被其中一个或所有相关组件拒绝。

计算机也能被网络事件从低电源级别中唤醒，以下情况会导致唤醒信号。

- ❒ 检测到网络链路状态的改变（例如，电缆重新连上）。
- ❒ 接收到网络唤醒帧。
- ❒ 接收到巨包（Magic Packet），巨包是包含连续 16 个接收方网络适配器介质访问（MAC）地址复制的数据包。

在驱动器初始化时，NDIS 查询微端口驱动器的能力以判定是否支持诸如巨包、模式匹配和链路状态改变唤醒等唤醒方式，并决定每种唤醒方式所要求的最低电源状态。然后，网络协议就只需查询微端口的能力，在运行时，协议设置使用对象标识符的唤醒策略，例如启用唤醒、设置包模式和删除包模式等。

目前，Windows TCP/IP 支持网络电源管理。它在微端口初始化时注册如下包模式。

- ❒ 直接 IP 包。
- ❒ 请求站 IP 地址的 ARP 广播。
- ❒ 请求站计算机名的 TCP/IP 上的广播。

与 NDIS 兼容的驱动程序适用于不同厂家的各种各样的网络适配器。NDIS 接口允许不同类型的多个协议驱动程序绑定到同一个网络适配器驱动程序，也允许将同一个协议绑定到多个网络适配器驱动程序上。NDIS 规范描述了实现这一点的多路复用机制。绑定可以通过 Windows 网络或拨号连接文件夹查看和改变。

Windows TCP/IP 对以下技术提供支持：光纤分布式数据接口（FDDI）、令牌环（IEEE802.5）、异步传输模式（ATM）。

链路层功能分布在网络适配器/驱动程序组合和低层协议栈驱动程序上。对 LAN 介质，网络适配器/驱动程序组合的过滤功能基于帧的目的 MAC 地址。正常情况下，LAN 硬件过滤掉目的地址不是以下地址之一的所有来帧。

- ❒ 适配器的单播 MAC 地址。
- ❒ 广播地址（对以太网，广播地址是 0xFFFFFFFFFFFF）。
- ❒ 通过协议驱动程序利用硬件注册的多播地址。

如果某个帧将这些地址之一作为其目的 MAC 地址，就能通过计算校验和来检查该帧的比特级完整性。所有通过了目的地址校验和检查的帧，都通过硬件中断提交给网络适配器驱动程序。网络适配器驱动程序是在计算机上运行的软件，因此，接收任何帧都需要一定的 CPU 处理时间。网络适配器驱动程序再通过接口卡把帧送入系统内存，然后再按组成帧时的顺序提交给特定的绑定传输驱动程序。NDIS5.0 规范提供了该过程的更多细节。

当一个包经过一个或一系列网络时，其源 MAC 地址是把该包放到传输介质上的网络适配器的 MAC 地址，而它的目的 MAC 地址总是通过该传输介质欲到达的网络适配器的 MAC 地址。这意味着在路由网络中，源和目的 MAC 地址在经过网络层设备（路由器或第三层交换机）的每一段时会改变。

16.1.2　IP 层

IP 层的功能是收发 IP 数据包，在 IP 层，除了实现 IP 协议外，还实现了 ARP、RARP 和 ICMP 等协议。下面分别进行介绍。

16.1.2.1　ARP

地址解析协议（ARP）为外出包进行 IP 地址到介质访问控制地址的解析。将外出数据报封装成帧时，必须填上其源和目的 MAC 地址。决定帧的目的 MAC 地址是 ARP 的任务。在 IP 路由选择过程中，一个外出 IP 数据报将选择接口（网络适配器）和转发 IP 地址。对外出 IP 数据报，ARP 将其转发 IP 地址与 ARP 高速缓存进行比较，以查找包将发往的网络适配器。如果有匹配项，就使用从高速缓存中得到的 MAC 地址，如果没有，ARP 就在本地子网上广播 ARP 请求帧，要求拥有所查询 IP 地址者回送它的 MAC 地址。当收到 ARP 响应时，就以新信息更新 ARP 高速缓存，并用它作为包的数据链路层地址。

Windows 能根据系统要求自动调整 ARP 高速缓存的大小。如果一个表项两分种内未被任何外出数据报使用，就从 ARP 高速缓存中删除掉。被访问过的表项赋以追加时间，每次增加 2 分钟，直到最大生命值时间 10 分钟。10 分钟后，就从 ARP 高速缓存中删除该表项，如果要继续使用，必须通过 ARP 请求帧重新查找。

除了通过接收 ARP 应答来创建 ARP 高速缓存表项外，也能根据从 ARP 请求中得到的映射信息来更新 ARP 表项。换句话说，如果 ARP 请求发送者的 IP 地址在高速缓存中，就用发送者的 MAC 地址更新表项。通过这种方法，含有发送者的静态或动态 ARP 高速缓存表项的结点，可以用发送者的当前 MAC 地址进行更新。接口或 MAC 地址发生改变的结点将更新 ARP 高速缓存，使其含有本结点下次发送 ARP 请求时要用到的表项。

在将目的 IP 地址解析成 MAC 地址时，ARP 仅能对一个外出 IP 数据报进行排队。如果基于 UDP 的应用程序连续地向同一目标地址发送多个 IP 数据报，某些数据报会因为不存在相应 ARP 高速缓存表项而被丢弃。

16.1.2.2　IP 路由

路由选择是 IP 层的基本功能。数据报经由网络适配器提交给 IP，每个数据报都有源和目的 IP 地址。IP 模块检查每个数据报的目的地址，并与本地维持的 IP 路由表作比较，以决定采取什么行动。对每个数据报，都有 3 种可能：提交给本地主机 IP 层之上的高层协议、通过本地某一网络适配器转发、丢弃。

Windows IP 路由表的每个表项包含如下信息。

- 目的网络：路由对应的网络 ID。目的网络可以是分类地址、子网、超网或主机路由的 IP 地址。
- 子网掩码：掩码用来匹配目的 IP 地址和目的网络。
- 网关：到目的网络的转发 IP 地址或下一路程段的 IP 地址。
- 接口：网络接口对应的 IP 地址，用于转发 IP 数据报。
- 度量：标示路由代价的数字，利用它可以在到达同一目的地的多条路由中选择一条最佳的。通常采用的度量是到目的网络的路程段数（经过的路由器数）。

如果两条路由有相同的目的网络和子网掩码，度量较小的路由就是最佳路由。路由表项

可用来存储以下类型的路由。

- 直接相连网络 ID 的路由：这些路由用于直接相连的网络 ID。对直接相连的网络而言，网关的 IP 地址就是该网络接口的 IP 地址。
- 远程网络 ID 的路由：这些路由用于那些不直接相连、但通过其他路由器可达的网络 ID。对远程网络而言，网关 IP 地址就是位于转发结点和远程网络之间的本地路由器的 IP 地址。
- 主机路由：面向特定 IP 地址的路由。主机路由允许在每个 IP 地址的基础上进行路由选择。对主机路由而言，目的网络就是特定主机的 IP 地址，子网掩码为 255.255.255.255。
- 默认路由：默认路由在未能找到特定网络 ID 或主机路由的情况下使用。默认路由的目的网络是 0.0.0.0，子网掩码为 0.0.0.0。

要决定一条转发 IP 数据报的路由，IP 模块使用如下过程。

- 对路由表中的每条路由，IP 模块将目的 IP 地址与子网掩码进行按位逻辑“与”操作，并将结果与目的网络进行匹配，如果匹配的话，IP 就将该路由标记为目的 IP 地址的匹配路由。
- IP 从所有匹配路由中选择子网掩码位数最多的路由。该路由与目的 IP 地址相匹配的位数最多，因而也是对该 IP 数据报最特殊的路由。这就是所谓寻找最长或最接近匹配的路由。
- 如果找到多条最接近匹配路由，IP 就选取度量最小的。
- 如果找到多条带最小度量的最接近匹配路由，IP 就从中随机选取一个。

在选定的路由上决定转发 IP 地址或下一路程段的 IP 地址时，IP 模块使用如下过程。

- 如果网关地址与接口地址相同，就将转发 IP 地址设为 IP 包的目的 IP 地址。
- 如果网关地址与接口地址不同，就将转发 IP 地址设为网关的 IP 地址。

路由决定过程的最终结果是从路由表中选定一条路由。路由选择产生一个转发 IP 地址（网关 IP 地址或 IP 数据报的目的 IP 地址）和一个接口（通过接口 IP 地址标识）。如果路由决定过程未能找到路由，IP 模块就宣告一个路由选择错误。对于发送主机，路由选择错误在内部被提交给 TCP 或 UDP 等上层协议。对于路由器，该 IP 数据报被丢弃并向源主机发送一个 ICMP“目的地不可达—主机不可达”消息。

Windows 提出了默认网关度量的新配置选项。该度量提供对任意时刻活动默认网关进行更好的控制。该度量默认值为 1，低度量路由比高度量路由好。在默认网关情况下，计算机使用度量最小的默认网关，除非该网关是不活动的，在这种情况下，死寂网关探测器把开关值切换到列表中下一个具有最小度量的默认网关。默认网关度量可以通过 TCP/IP 高级配置选项进行设置。DHCP 服务器能提供一个基本度量和一组默认网关。如果 DHCP 服务器提供的基本度量为 100，并提供 3 个默认网关，那么这三个网关的度量分别为 100、101 和 102。DHCP 提供的基本度量不适用于静态配置的默认网关。

大多数自治系统（AS）路由器用路由选择信息协议（RIP）或开放最短路径优先（OSPF）同其他路由器交换路由表信息。Windows 以路由选择和远程访问服务支持这些协议。Windows 也支持沉默的 RIP，方法是使用 RIP 进行侦听，该功能是可选的网络服务。默认情况下，基于 Windows 的系统并不像路由器那样工作，也不在接口间转发 IP 数据报。路由选择和远程

访问服务包含在 Windows 中，可以启用并通过配置提供完全的多协议路由选择　服务。

16.1.2.3　重复 IP 地址检测

重复 IP 地址检测保证一个 IP 结点所使用的 IP 地址在其所连接的网段上是惟一的。当协议栈第一次初始化时，Windows 对主机自身 IP 地址发送 ARP 请求包解析自己的 IP 地址，称之为伴随 ARP。如果任何其他主机响应了这样的 ARP 请求，该 IP 地址就已经被占用了。ARP 高速缓存项在收到 ARP 请求后就会更新。因而，在向占用地址系统发送了单播 ARP 应答后，被占用地址系统会广播一个附加的伴随 ARP 请求，以便网络中其他主机能在它们的 ARP 高速缓存中维持正确的地址映射。当计算机不与网络相连时，用户可以用重复的 IP 地址启动它，在这种情况下，不会检测到冲突。但是，如果用户随后把它加入到网络中，当它第一次对其他 IP 地址发送 ARP 请求时，任何使用该冲突地址的 Windows 计算机将检测到冲突并保持运行。如果两台计算机都运行 Windows，IP 在两台有重复地址的计算机上都保持运行。检测到冲突的计算机将显示一条出错消息并在系统日志中产生详细的日志。Windows DHCP 允许客户机执行重复 IP 地址检测，条件是客户机进入 DHCP 选择状态。如果检测到重复 IP 地址，DHCP 客户机就向 DHCP 服务器发送一个 DHCP 拒绝数据包，然后进入 DHCP 初始化状态。在收到 DHCP 拒绝数据包后，DHCP 服务器就将该 IP 地址置为不可用。

16.1.3　传输层

16.1.3.1　TCP

传输控制协议（TCP）为应用程序提供基于连接的、可靠的字节流服务。Windows 网络依靠 TCP 来实现登录过程、文件和打印共享、域控制器之间的信息复制、浏览列表传输和其他常用功能。它也能用于一对一的通信。TCP 使用检验和来检查 TCP 报头和 TCP 段中有效载荷的传输错误，以减少网络出错而未被检测到的概率。

TCP 接收窗口大小的计算和窗口缩放。TCP 接收窗口大小是指每次在一个连接上能缓存的接收数据量（以字节计）。在接收主机等待应答数据并更新窗口之前，发送主机只能发送这么多数据。Windows TCP/IP 设计成能在大多数情况下进行自我调节，从而能比以前版本使用更大的默认窗口。TCP 的窗口大小能适应连接建立期间所协商的最大段长（MSS）的平缓增加，而不是采用硬编码的默认窗口大小。使接收窗口能适应 MSS 的平缓增加提高了大批数据传输过程中使用的满载 TCP 段的比例。默认情况下接收窗口大小按如下方式计算。

（1）发往远程主机的第一个连接请求通告一个接收窗口大小，一般为 16 千字节（KB），即 16 384 字节。

（2）一旦建立了连接，接收窗口大小就舍入成连接建立期间所协商的 TCP 最大段长（MSS）的整数倍。

（3）如果舍入值不到 MSS 的 4 倍，就把它调整到 4×MSS，同时最大值限制为 64KB，除非窗口缩放选项被启用。

（4）基于以太网的 TCP 连接，其窗口大小正常时为 17 520 字节，或舍入到 16KB（即 12 个 1460 字节的字段）。

为了提高高带宽、高延迟网络的性能，Windows TCP 支持 RFC1323 中定义的 TCP 窗口缩放。通过在 TCP 三次握手期间商定一个窗口缩放因子，支持 TCP 接收窗口的大小可大于



64KB。它支持的接收窗口最大可达 1GB。在阅读支持可变窗口主机之间所建立的连接的有关信息时，必须记住段中通告的窗口大小要乘以商定的缩放因子。窗口缩放因子只在三次握手的前两个段中出现。缩放因子是 2^S，其中 S 为商定缩放因子。例如，对缩放因子为 3 的通告窗口大小 65 535，实际接收窗口大小为 524 280 即 $2^3 \times 65\ 535$。

TCP 采用延迟应答来减少传输介质中的包数。Windows TCP 采用变通的方法实现延迟 ACK，并不对每个接收到的 TCP 段发送应答。TCP 在给定连接上收到数据，只有当以下条件符合时才回送应答。

- 没有为以前接收到的段发送过 ACK。
- 接收到一个段，但在该连接上 200ms 之内没有接收到其他段。
- 通常为连接上接收到的每个其他 TCP 段都发送 ACK，直到延迟 ACK 定时器（200ms）超时。

Windows 开始支持一项称为选择性应答（SACK）的重要性能特性。SACK 对使用较大 TCP 窗口的连接很重要。在 SACK 之前，接收者只能应答连接上接收到的最后一个数据，或接收窗口的左边界。当 SACK 启用时，接收者连续地使用 ACK 编号来应答接收窗口的左边界，但接收者也能对不连续的接收数据块单独进行应答。SACK 在 TCP 连接建立期间用 TCP 报头选项来协商 SACK 的使用，并标明接收数据块的左右边界。可以指示多个接收块。默认情况下，SACK 是启用的。当一个段或一系列段以非连续模式到达时，接收者能准确地通知发送者哪个数据已收到，也隐含表明哪个数据没到达。发送者能够有选择地重发所缺数据，而不必重发已成功接收的整个数据块。

TCP 重发行为。当每个出站段传递给 IP 时，TCP 都启动一个重发定时器。如果在定时器超时前没收到给定段中数据的应答，就重发该段。对于新的连接请求，重发定时器初始化为 3 秒。重发超时（RTO）在外出段的基础上进行调整，以匹配使用平滑往返时间（SRTT）计算的连接特性和 Karn 算法。给定段的定时器值在每次重发该段后就加倍。采用这种算法，TCP 自己就能适应连接的“正常”延迟，高延迟链路上的 TCP 连接比低延迟链路上的要经历更长的时间才超时。

在某些情况下无须重发定时器超时就需进行重发。最常见的一种情况称为快速重发。如果支持快速重发的接收者接收到的数据中所包含的序列号超过期望值，某些数据很可能丢失了。为帮助发送者意识到这件事，接收者立即发送 ACK，其应答序号设为所期望的序列号。对到来数据流中每个在丢失数据之后的段，接收者同样处理。当发送者开始接收到一系列应答同一序列号的 ACK，并且该序列号比当前发送的序列号小时，它就能推断出某一（某些）段已丢失了。支持快速重发算法的发送者立即发送接收者所期望的段，接收者将用这些段来填充接收数据中的缺口，而不必等到该段重发定时器超时。这种优化在高丢失率网络环境中极大地提高了性能。

TCP 保持活动消息。TCP 保持活动包只是一个简单的 ACK，其序列号比本连接的当前序列号小 1。主机接收到这种 ACK 后，就以当前序列号应答。保持活动可用来证实本连接的远程计算机仍是可用的。TCP 保持活动包每个时间段内发送一次，时间段的长短由 KeepAliveTime 的值（默认为 7 200 000ms，即 2h）决定，前提是没有其他数据或更高级别的保持活动包在本连接中传输。如果保持活动包没被应答，就在每个时间段重发一次，时间段的长度是与 KeepAliveInterval 的值相等的秒数。默认时 KeepAliveInterval 的值为 1s。在 NetBT

连接中，正如许多 Windows 联网组件所使用的那样，以更高的频率发送 NetBIOS 保持活动数据包。因此，在 NetBIOS 连接上就不再发送正常的 TCP 保持活动包。TCP 保持活动功能默认时是关闭的。

慢启动算法和拥塞避免。Windows TCP 支持慢启动和拥塞避免算法。一个连接建立后，TCP 起先只缓慢地发送数据来估计连接的带宽，以避免淹没接收主机或通路上的其他设备和链路。发送窗口大小设为 2 个 TCP 段，当两个段都被应答后，窗口大小就扩大为三个段。当三个段都被应答后，发送窗口大小再次扩大。如此进行，直到每次突发传输的数据量达到远程主机所声明的接收窗口大小。此时，慢启动算法就不再用了，改用声明的接收窗口进行流控制。在传输中的任何时刻都可能发生拥塞。当重发定时器超时，或接收到说明已经有 TCP 段被路由器丢弃的 ICMP“源中止”消息时，就可检测到拥塞。发生这种情况时，TCP 拥塞避免算法就减小发送窗口的大小，并使其逐步减小到拥塞发生时的窗口大小的一半，然后，使用慢启动算法来增大发送窗口，使其达到接收主机接收窗口的大小。

糊涂窗口综合症。糊涂窗口综合症（SWS）指声明的接收窗口小于一个完整的 TCP 段。糊涂窗口综合症会导致发送很小的 TCP 段，致使网络使用效率非常低。Windows TCP/IP 实现了 RFC1122 中描述的发送者和接收者 SWS 避免。接收端 SWS 避免的实现是：在增加的数据小于一个 TCP 段之前不打开接收窗口。发送端 SWS 避免的实现是：在接收端声明发送完整 TCP 段的有效窗口大小之前不发送更多数据。发送端 SWS 避免有例外情况，参见 RFC1122。

TCP TIME-OUT 延时。TCP 连接关闭后，该连接就进入一种称为 TIME-OUT 的状态，以确保新连接不会使用相同的协议、源 IP 地址、目的 IP 地址、源端口和目的端口，直到经过了足够长的时间，能确定不会有任何被错误路由或延迟的段突然出现。套接字对不应当被再次使用的时间长度由 RFC793 定义，一般为最大生命周期的两倍（2MSL）或 240s（4min），这是 Windows 的默认值，但是，在使用默认值时，某些在短时间内执行大量出站连接的应用程序可能在端口能被重用之前耗尽所有的可用端口。Windows 对这种情况提供两种控制方法。

第一种：用注册表表项 TcpTimedWaitDelay（HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters）来改变这个时间值。Windows 允许将该值设成只有 30s，这在大多数情况下就不会有问题了。

第二种：通过注册表表项 MaxUserPort（HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters）来配置源出连接的用户可访问的临时端口数。默认情况下，当应用程序为出站调用申请任何套接字时，将采用端口号在 1024～5000 的端口。用户可通过注册表表项 MaxUserPort 设置出站连接可使用的最高端口号。例如，将值设为 10 000 将有大约 9000 个可供出站连接使用的用户端口。

吞吐量因素。Windows TCP/IP 在大多数网络条件下都适用，它能为每个连接动态地提供可能的最大吞吐量和最佳可靠性。TCP 设计成能在不同的链路条件下提供最优性能。一个链路上的实际吞吐量与很多因素有关，最主要的有以下几个：链路速度（每秒可传输的比特数）、传播延迟、窗口大小（TCP 连接上可发送的未响应数据量）、链路可靠性、网络和中间设备拥塞。

吞吐量的关键因素包括：



通信信道（又称管道）的容量，即带宽延迟量，是往返时间乘以带宽（比特率）。如果某一链路的比特级错误很少，最佳性能时的窗口大小应大于或等于带宽延迟量，以使发送者能充满管道。不启用窗口缩放时，可用的最大窗口为 65 535，因为窗口域在 TCP 报头中只占 16 位。启用窗口缩放时，窗口大小可达 1GB。

吞吐量决不能超过窗口大小除以往返时间。

如果链路有很多比特级错误或经常严重拥塞以致于要丢弃包，使用更大的窗口就不能提高性能。Windows 支持 SACK，以改善高丢失率环境下的性能，也支持 TCP 时标，以改善 RTT 估计。

传播延迟取决于向不同传输方向传送光或电信号的延迟以及传输设备和中间系统的延迟。

传输延迟取决于传输介质的速度和介质访问控制模式的本质属性。

对特定路径，传播延迟是固定的，但传输延迟取决于包大小和拥塞情况。

低速时，传输延迟是限制因素；高速时，传播延迟就可能成为限制因素。

16.1.3.2 UDP

用户数据报协议（UDP）提供无连接、不可靠的传输服务。它通常用于使用广播或多播 IP 数据报的一到多通信。由于 UDP 数据报的传送是得不到保证的，所以使用 UDP 的应用程序必须通过简单的重发或其他可靠性机制来补偿丢失的 UDP 数据报。Windows 网络使用 UDP 进行登录、浏览和 NetBIOS 名字解析。

UDP 可用于 NetBIOS 名字解析，方法是通过 NetBIOS 名字服务器单播或子网广播，也可用于将域名系统（DNS）主机名解析成 IP 地址。NetBIOS 名字解析由 UDP 端口 137 完成，DNS 查询使用 UDP 端口 53。由于 UDP 本身不保证数据报的传送，这两种服务在收不到查询回答时，就使用自己的重发机制。UDP 广播数据报一般不用 IP 路由器转发，因此路由环境下的 NetBIOS 名字解析需要 WindowsInternet 名字服务（WINS）之类的名字服务器，或者使用静态数据库文件，如 Lmhost 文件。

16.1.4 TCP/IP 开发接口

Windows 网络应用程序可采用多种方法通过 TCP/IP 协议栈进行通信。有些方法如命名管道等要通过网络重定向器，它是工作站服务的一部分。许多老的应用程序是根据 NetBIOS 接口编写的，由 TCP/IP 上的 NetBIOS 支持。但这里只简单说明 Windows 套接字接口。

Windows 套接字定义了一个编程接口，该接口基于加利福尼亚大学伯克利分校的套接字接口。它还包括一组扩展设计以充分利用 Windows 的消息驱动特性。该规范的 1.1 版本于 1993 年 1 月发布，2.2.0 版本于 1996 年 5 月发布，Windows 2000 支持版本 2.2，通常又称 Winsock2。

有很多可用的 Windows 套接字应用程序。Windows 中就包含大量的基于 Windows 套接字的应用程序，如 Ping 和 Tracert 工具、FTP 和 DHCP 客户机及服务器，以及 Telnet 客户。也有很多基于 Winsock 的高级编程接口。

名字和地址解析。Windows 套接字应用程序一般使用 gethostbyname 函数把主机名解析成 IP 地址。默认情况下，gethostbyname 函数采用以下的名字查找步骤。

（1）检查请求的名字是否与本主机名匹配。

（2）检查主机文件，看是否有匹配的名字。

（3）如果配置了 DNS 服务器，就查询它。

（4）如果没找到匹配项，进行 NetBIOS 名字解析过程，直至进行 DNS 名字解析。

某些应用程序使用 gethostbyaddr 函数把 IP 地址解析成主机名。gethostbyaddr 调用执行以下动作（默认）时：

（1）检查主机文件，以寻找匹配地址项。

（2）如果配置有 DNS 服务器，就询问它。

（3）向被查询的 IP 地址发送一个 NetBIOS 适配器状态请求，如果它已注册给该适配器的一组 NetBIOS 名字进行应答，就从中分析计算机名字。

Winsock2 支持 IP 多播。但目前只有 IP 族数据报和原始套接字支持 IP 多播。

保留值参数。Windows 套接字服务器应用程序通常创建一个套接字，然后在该套接字上用 listen 函数接收连接请求。传递给 listen 函数的参数之一是保留值参数，说明应用程序希望 Windows 套接字为本套接字排队的连接请求数。

16.2　UNIX/Linux 的 TCP/IP 实现

Linux 是开放源码的操作系统，它具有强大的网络功能。Linux 的网络实现是以 4.3BSD 为模型的，它支持 BSD Sockets（及一些扩展）和所有的 TCP/IP 网络。选这个编程接口是因为它很流行并且有助于应用程序从 Linux 平台移植到其 Unix 平台。因此可以看出，虽然 UNIX 操作系统和 Linux 操作系统存在着一定的区别，但它们的系统结构和大多的实现技术都是相同的。在 TCP/IP 协议的实现上，两者也大致相同。所以本节主要以 Linux 操作系统中的 TCI/IP 协议的实现为基础说明 UNIX/Linux 操作系统中的 TCP/IP 协议的实现机制。

在 UNIX 和 Linux 系统中，协议栈的实现通常都采用 BSD 或 STREAMS 两种结构之一。从概念上看，STREAMS 结构是一种模块化的系统结构，具有很好的灵活性和扩展性，而 BSD 结构是一种分层结构。图 16.2 说明了两种结构的区别。

从图 16.2 可以看出，BSD 结构比较简单，但扩展性不如 STREAMS 结构。STREAMS 结构可以在同一个系统中同时实现多种传输协议而提供统一的结构和对上层的接口。

16.2.1　Linux 网络协议栈

Linux 中，协议栈是作为内核的一部分实现的，因此它包含在内核代码中，如图 16.3 所示。

从图 16.3 可以看出，Linux 的协议栈可以支持多种协议。在物理链路层，除了支持以太网外，还可以支持帧中继、FDDI 等。而在传输层除了 TCP/IP 协议族中的 UDP 和 TCP，还可以支持 AppleTalk、IPX 等传输层协议。尽管 Linux 对多协议的支持，本书中只讨论与 TCP/IP 协议相关的部分。

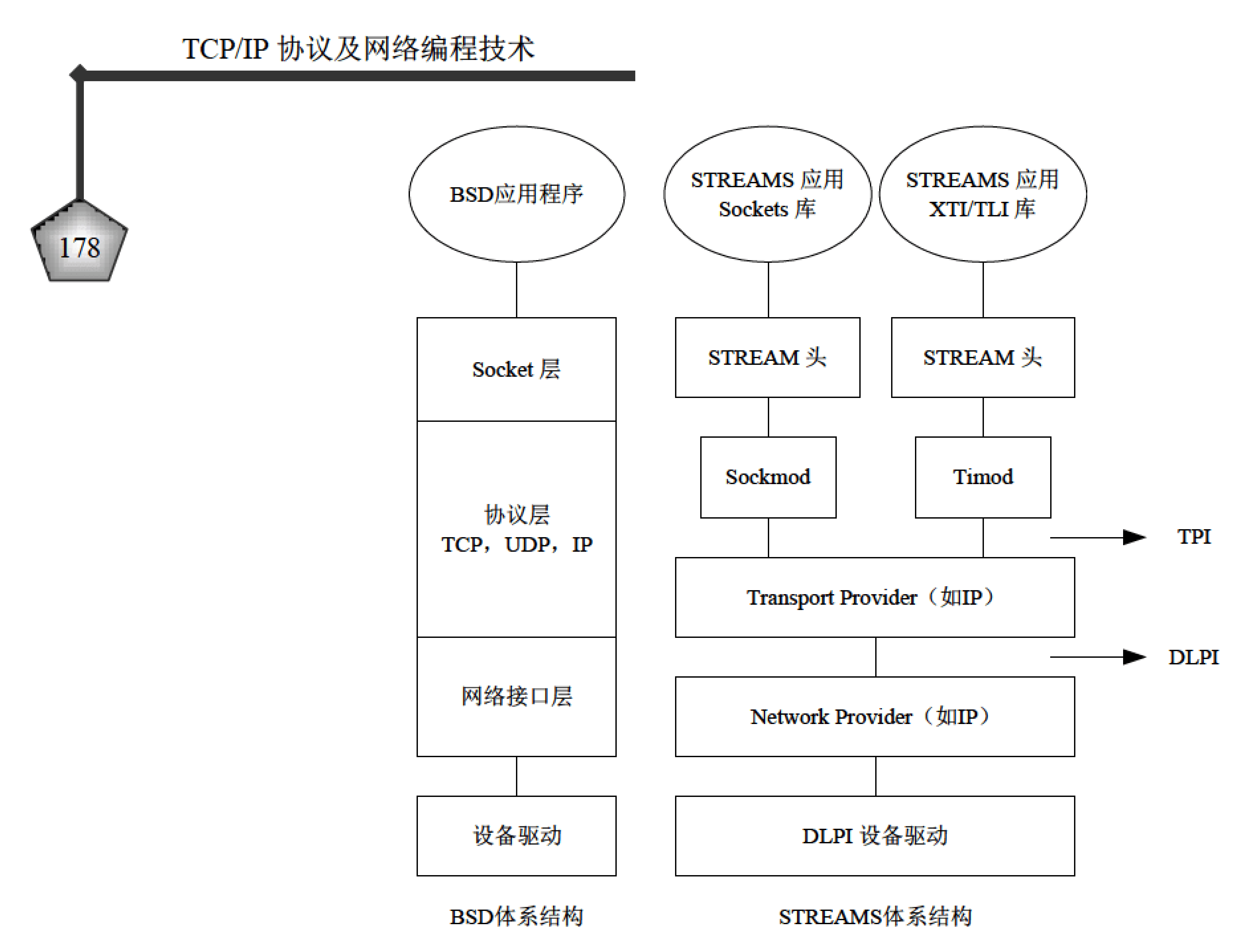


图 16.2　BSD 协议栈结构和 STREAMS 协议栈结构

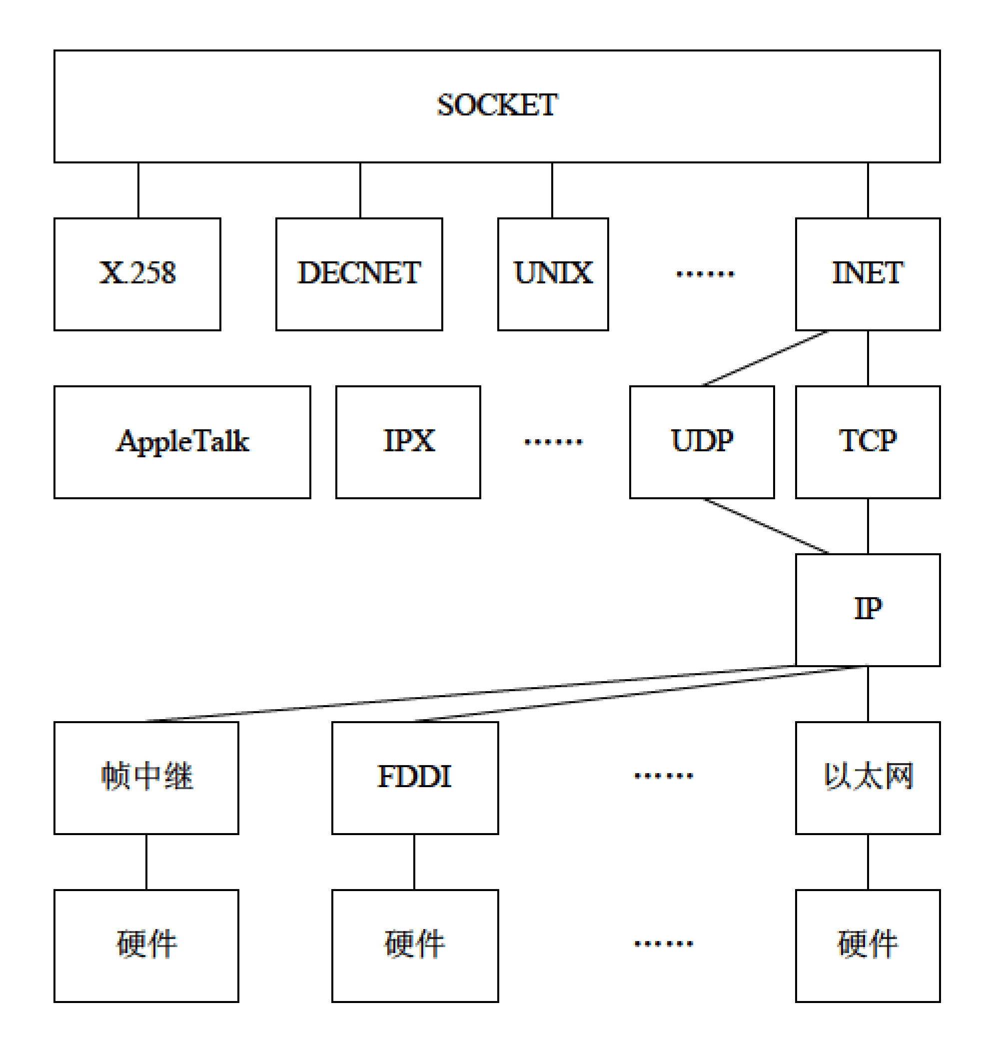


图 16.3　Linux 协议栈结构

16.2.2　Linux 网络数据处理流程

基于上述的 Linux 中严格的分层实现体系，在 Linux 系统中，一个应用程序发送/接收数据的流程如图 16.4 所示。

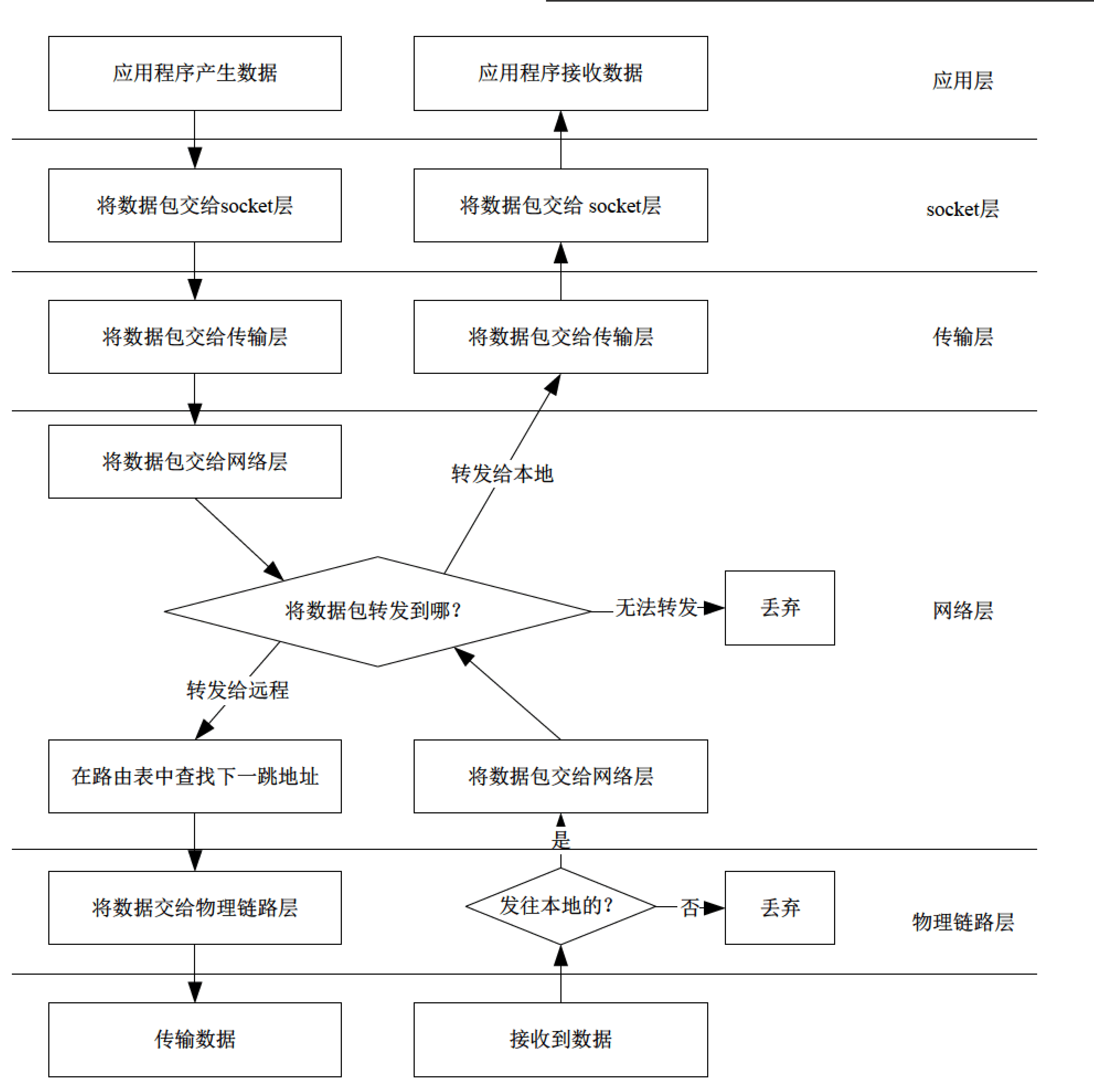


图 16.4　Linux 中网络数据处理流程

使用分层结构实现协议栈即符合 TCP/IP 协议的分层思想，又利于协议栈的灵活性和可扩展性，又使得协议数据可以按分层进行处理。图 16.4 中，数据的发送是按应用层、socket 层、传输层、网络层、数据链路层从上到下的顺序进行分层处理的，而接收到的数据则正好是按照相反的顺序从下向上进行处理的。

应用程序产生数据后，就将其通过 socket 提交给传输层（TCP 或 UDP），传输层将其交给网络层处理。在网络层，Linux 内核会在路由缓存或转发信息库中查找路由信息。如果数据包是发给其他计算机的，内核就会将其交给链路层输出到网络接口，并最终发送到物理传输介质上。

当一个数据包从物理介质到达输入接口时，它会检查该数据包是否确实是发往该计算机的。如果是，它就将其交给 IP 层，IP 层查找路由表。如果该数据包是发往其他计算机的，IP 层就将它向下交给输出接口。如果该数据包是发往本地计算机的上层应用程序的，它就将其通过传输层和 socket 层交给应用程序进行处理。

16.2.3 Linux 的 IP 路由

从图 16.4 可以看出，IP 层在数据的发送和接收过程中起着关键的作用，它能够判断一个数据包应该通过那条路径到达它的目的地。因为 Linux 系统可以是一个主机系统或者是一个路由器，所以 IP 层的路由表的具体实现和查找机制在数据发送和接收过程中显得非常 重要。

当 Linux 的内核装载时，它会读取一系列的配置文件并执行一系列的任务，其中就包括建立计算机的网络连接的过程。该过程会配置好计算机的地址，初始化网络接口，建立好路由表并向其中添加静态路由。

整个的配置过程可以是动态的，也可以是静态的。配置地址的方式可以有两种，如果计算机有一个固定的地址，系统管理员可以在配置文件中指定。那么系统启动时，计算机的地址就是该地址了。如果计算机没有固定的地址，主机可以使用 DHCP 协议来从 DHCP 服务器获得计算机的地址，路由器和 DNS 服务器等信息。

Linux 中使用两种复杂的路由表来维护转发信息：转发信息表（FIB）用来保存所有可能的转发地址，路由缓存比 FIB 小但查找速度快，是经常使用的路由表。一个 IP 数据包需要发送到远程主机时，IP 层首先在路由缓存中查找合适的项。如果找到，IP 就使用它进行数据包的转发。如果没找到，IP 就从 FIB 中进行查找，并将找到的表项添加到路由缓存中，然后使用该表项进行数据包的转发。

根据网络状态的变化，路由表需要进行相应的改变。Linux 中路由缓存会经常变化，但 FIB 几乎是静态的，只有在网络状态发生变化时才会发生改变。

第 17 章　标准 TCP/IP 编程接口——Socket

从本章开始，我们介绍如何使用套接口 API 开发网络应用程序。大部分网络应用系统都可以分为客户端（Client）和服务器端（Server），也就是通常所说的 C/S 结构。两者相比较，客户端开发需要解决的是用户界面的友好性，服务器开发则需要解决设计上的健壮性和扩展性。友好的界面固然重要，但并不是网络应用考虑的重点，我们的最终目标是让读者能够开发一个 Windows 平台下的具有相当扩展性和健壮性的实用服务器程序。需要说明的是，本书并不是 Winsock 的用户编程指南，它只涉及到我们认为读者必须掌握的那些 API 函数。虽然讨论的是 Windows 平台中的网络编程，但其中的很多技术及思想可以用于其他操作系统中。

我们的测试环境是 Windows 2000 Server（SP4）系统，VC6.0+SP5+Platform Core SDK2003。书中的所有源码均在该环境下编写并测试通过。

17.1　套接口概述

在互联网协议中两种常用的应用编程接口是套接口（Sockets）和运输层接口（TLI）。因为第一个被广泛使用的 TCP/IP 协议栈和套接口 API 版本是在 4.2BSD 系统（1983 年）中发布的，前者也常被称为伯克利套接口（Berkeley Sockets）。目前它已被广泛地移植到很多非 BSD UNIX 系统和非 UNIX 系统中，其中就包括 Windows。后者最初由 AT&T 开发，由于被 X/Open（即现在的 Open Group）承认，有时也叫做 XTI（X/Open 传输接口）。X/Open 开始时是一个由欧洲、美国和亚洲的国际 UNIX 厂家组成的协会，现在已经成为像 POSIX 和 ANSI 一样的标准化组织之一。

本书主要介绍基于套接口 API 的网络编程，不涉及到 TLI，对后者感兴趣的读者可以参阅 W. Richard Stevens 关于网络编程的经典读物《UNIX 网络编程（第一卷）——套接口 API 和 X/Open 传输接口 API》。

Berkeley Sockets 接口在 Windows 平台上的移植版本称为 Winsock，它不仅包含前者的大部分函数，还包含一组针对 Windows 系统的扩展库函数（通常以字母 WSA 打头），这些函数使编程人员能充分利用 Windows 的消息机制以及 Win32 平台下的高性能 I/O 模型。最初的伯克利套接口 API 在 Windows 平台下的移植版本是 Winsock1.1，在它的基础上，微软又进一步提供了 Winsock2 接口。在 Winsock 1.1 版本中，不同的 TCP/IP 协议栈供应商需要提供自己的 Winsock 接口实现的动态链接库，因此底层协议栈与 Winsock 的接口在不同实现之间是完全不同的。为了能同时提供对多种底层网络传输协议访问的能力，Winsock2 的框架结构

与 Winsock 1.1 版本相比做了很大的改进。它在 Winsock 接口与协议栈之间定义了一种标准服务提供接口（SPI），这就使得同一个 Winsock 动态链接库能同时访问多个由不同供应商提供的协议栈。并且，协议栈供应商不再需要提供自己的 Winsock 接口实现链接库，微软提供了单独的 WS2_32.dll，该链接库已经成为 Windows 系统的一个基本组件。

Winsock2 框架结构如图 17.1 所示，在该图中有 3 点值得注意：首先，在编写 Winsock 应用程序时，一般并不直接使用 Ws2_32.dll，而是用其对应的静态链接库 Ws2_32.lib；其次，Winsock 内核模式驱动 afd.sys 是 Winsock2 网络缓存的管理器，应用程序是从 afd.sys 的内部缓冲区复制网络数据，而发送数据事实上也是向该内部缓冲区复制数据（后文将提到如何对该缓冲区的大小进行控制）；最后，Winsock2 支持多种底层的网络协议，如 TCP/IP、ATM 等，本书只涉及到 TCP/IP 部分。

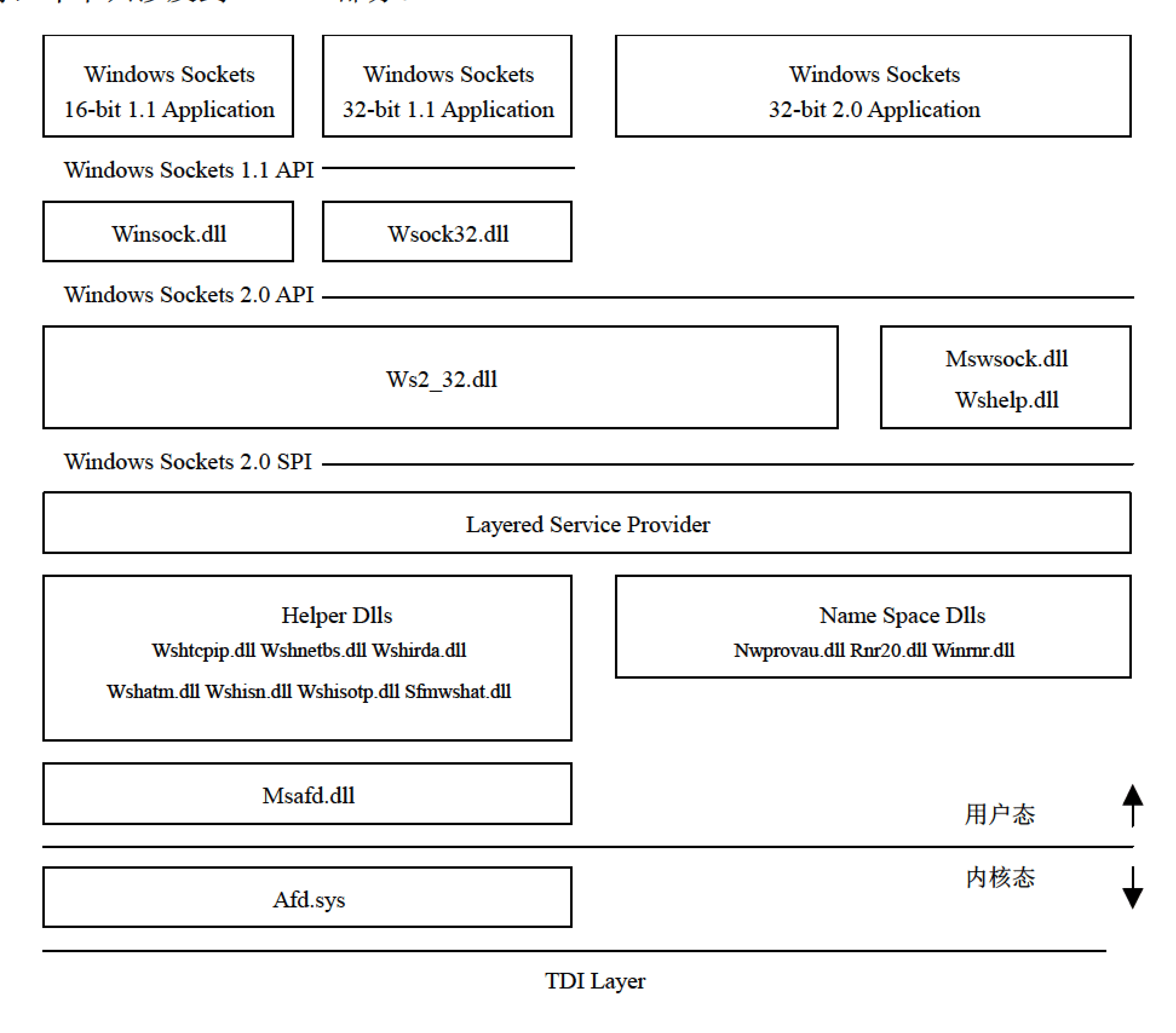


图 17.1　Winsock2 框架结构

那么，从用户的角度来看，套接口究竟是什么呢？事实上，套接口是网络通信端点的一种抽象概念，它为用户提供了一种发送和接收数据的机制。如果需要在分布于不同主机上的进程之间进行通信，那么 Sockets 是一种理想的手段，我们可以通过它来交换数据。因此，Sockets 是一种进程间通信的机制，适用于分布式环境。

从下一节开始，我们将逐步介绍 Winsock 编程的一些基本概念和函数。



17.2　地址与地址操作函数

要实现通信，就必须要有地址的概念。在 Socket 编程过程中，经常会碰到 3 种类型的套接口地址结构，分别是 INET 协议族地址结构、IPv4 地址结构和通用地址结构。

17.2.1　INET 协议族地址结构——sockaddr_in

地址结构名中的最后两个字母“in”，是 Internet 的简写，说明该结构仅适用于采用 TCP/IP 协议的网络，结构定义如下：

```
/* Socket地址，internet类型 */
struct sockaddr_in {
        short           sin_family;
        u_short         sin_port;
        struct in_addr  sin_addr;
        char            sin_zero[8];
};
```

- sin_family：地址族，一般填为 AF_INET。在网络编程过程中，有可能会碰到另一组和 AF_XXX 类似的 PF_XXX 常量，与 AF_INET 相对应有 PF_INET。历史上，PF_XXX 被设计用于表示协议族，而 AF_XXX 用于表示地址族。最初的设想是单个协议族可以支持多个地址族，PF_XXX 用于套接口的创建，AF_XXX 用于套接口地址结构。但事实上，这种设计思想从来没有成为现实。在 Winsock2.h 文件中可以发现，PF_XXX 被定义为与 AF_XXX 值完全相同。因此，在本书中，统一使用 AF_XXX 常量定义。
- sin_port：16 位的 IP 端口，必须注意字节序问题，见 17.4 节。
- sin_addr：32 位的 IPv4 地址，见 17.2.2 节。
- sin_zero：8 个字节的 0 值填充，惟一的作用是使 sockaddr_in 结构大小与通用地址结构 sockaddr（见 17.2.3 节）相同。虽然很少有函数要求该结构域必须为零，但是在给结构体赋值之前先将其全部初始化为零是一个好习惯。下面的两个函数经常被用来完成清零工作，由于前者仅适用于 Win32 平台，推荐使用后者。

```
VOID ZeroMemory(PVOID destination, SIZE_T length);
void *memset( void *dest, int c, size_t count );
```

17.2.2　IPv4 地址结构——in_addr

用于存储 32 位 IPv4 地址的数据结构，其定义如下：

```
/* Internet address (old style... should be updated) */
struct in_addr {
```

```
        union {
              struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
              struct { u_short s_w1,s_w2; } S_un_w;
              u_long S_addr;
        } S_un;
#define s_addr      S_un.S_addr     /* can be used for most tcp & ip code */
#define s_host      S_un.S_un_b.s_b2    /* host on imp */
#define s_net       S_un.S_un_b.s_b1    /* network */
#define s_imp       S_un.S_un_w.s_w2    /* imp */
#define s_impno     S_un.S_un_b.s_b4    /* imp # */
#define s_lh    S_un.S_un_b.s_b3    /* logical host */
};
```

该结构提供了 3 种赋值的接口 S_addr、S_un_b 和 S_un_w，最常用的是前两种。

- S_addr：32 位的无符号整数，对应 32 位 IPv4 地址。要将地址 202.119.9.199 赋给 in_addr 结构，可以使用如下代码：

```
in_addr addr;
addr.S_un.S_addr = inet_addr("202.119.9.199");
```

其中，inet_addr 函数用于转换点串 IP 地址，将在 17.2.4 节进行详细介绍。

由于有定义：

```
#define s_addr  S_un.S_addr    /* can be used for most tcp & ip code */
```

故也可以将上面的代码简写为：

```
in_addr addr;
addr.s_addr = inet_addr("202.119.9.199");
```

假设主机上有多块以太网卡，每块网卡都配有 IP 地址，并且不关心应用程序具体使用哪个接口，那么在给 addr.s_addr 赋值时可以使用常量——INADDR_ANY。它在 Winsock2.h 中被定义为(u_long)0x00000000，即本地的任意以太网接口 IP 地址。代码如下：

```
in_addr addr;
addr.s_addr = INADDR_ANY;
```

- S_un_b：包含 4 个 8 位无符号整数，组合起来表示 IPv4 地址：s_b1.s_b2.s_b3.s_b4。下面的例子同样将 IPv4 地址 202.119.9.199 赋给 addr。

```
in_addr addr;
addr.S_un.S_un_b.s_b1 = 202;
addr.S_un.S_un_b.s_b2 = 119;
addr.S_un.S_un_b.s_b3 = 9;
addr.S_un.S_un_b.s_b4 = 199;
```

17.2.3　通用地址结构——sockaddr

```
/* Structure used by kernel to store most addresses */
struct sockaddr {
        u_short  sa_family;          /* address family */
        char     sa_data[14];        /* up to 14 bytes of direct address */
};
```

许多编程人员可能会对通用地址结构的存在感到迷惑。事实上，由于网络底层协议的多样性，研究人员在最初设计套接口函数接口时，面临着这样的选择：是专门开发一套为 TCP/IP 协议所用的 API，还是提供一种通用的编程接口以服务于多种网络协议。两者之间的差别非常明显，如果是采用前者，那么提供的函数接口就会相对简单，而对于后者，程序员在使用时必须提供足够的信息（参数）来告诉接口自己所采用的协议族。以 connect 函数为例（该函数一般用于主动建立 TCP 连接）：

```
int connect(SOCKET s, const struct sockaddr FAR *name, int namelen);
```

为了使其适用于不同的网络协议环境，它的第二个参数并不是 struct sockadr_in *，而是 struct sockaddr *。在使用涉及到这种地址结构的函数接口时，必须强制将 struct sockadr_in 指针转化为 struct sockaddr 指针。很多人会奇怪为何不采用 ANSI C 的通用指针类型 void *，答案很简单：套接口函数接口是在 ANSI C 标准制定之前定义的。

对于程序员来说，很少需要直接使用通用地址结构，惟一需要记住的是务必进行强制地址转换。

17.2.4　地址操作函数

本节介绍 3 个常用的地址操作函数 inet_addr、inet_ntoa 以及 gethostbyname。这里需要强调的是，在使用 Winsock 函数之前，应用程序必须首先调用 WSAStartup 函数初始化 ws2_32.dll，而在应用结束后必须调用 WSACleanup 函数，这两个函数的详细介绍见 17.6.1 节和 17.6.10 节。

1．函数 inet_addr 将包含点分格式的 IPv4 地址字符串转化为 in_addr 地址结构适用的 32 位整数。其定义如下：

```
unsigned long inet_addr(const char FAR *cp);
```

- cp：[IN]*，NULL 结尾的点分 IPv4 字符串。
- 返回值：如果没有错误发生，函数返回 32 位的地址信息。如果 cp 字符串包含的不是合法的 IP 地址，那么函数返回 INADDR_NONE。

2．与 inet_addr 相反，函数 inet_ntoa 将一个 in_addr 地址值转化为标准的点分 IP 地址字符串。定义如下：

```
char FAR * inet_ntoa(struct in_addr in);
```

* 此处[IN]表示该参数为输入参数，此外[OUT]表示输出参数，[INOUT]表示输入输出参数。



- in：[IN]，IPv4 地址结构。
- 返回值：如果没有错误发生，函数 inet_ntoa 返回一指向包含点分 IP 地址的静态存储区字符指针；否则返回 NULL。
- 注释：保存在该指针指向的存储区中的信息仅确保在下一次 Winsock 调用之前有效，因此应该及时加以复制。

最后，用一个简单的例子来结束对套接口地址结构的介绍，下面的代码片断演示了如何为 INET 地址结构赋值，并在 connect 函数中使用该地址。再次强调，其中的 WSAStartup 函数用于初始化 Winsock，它是所有 Winsock 应用程序在使用 Winsock 库之前都必须调用的；socket 函数用于创建套接口；connect 函数用于连接服务器；WSACleanup 函数用于结束 Winsock 的使用。这些函数的使用在后文中都将加以详细介绍。

```
WSAStartup(…);
……
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in to;
memset(&to, 0, sizeof(to));
to.sin_addr.s_addr = inet_addr("202.119.9.199");
to.sin_family = AF_INET;
to.sin_port = htons(5555);
connect(sock, (struct sockaddr *) &to, sizeof(to));
……
WSACleanup();
```

3．与上述两个函数不同，gethostbyname 完成的是域名解析功能。函数定义如下：

```
struct hostent FAR *gethostbyname(const char FAR *name);
```

- name：[IN]，待解析的 NULL 结尾的域名字符串。
- 返回值：如果没有错误发生，函数返回包含域名地址信息的 HOSTENT 结构数据；否则返回空指针（NULL），可以调用 17.6.1 节中介绍的 WSAGetLastError 函数来获得具体的错误码。在 HOSTENT 结构中我们最感兴趣的是 h_addr_list 域，它是一个 NULL 结尾的 IP 地址列表。
- 示例：下面的代码完成对www.seu.edu.cn的域名解析，并输出获得的 IP 地址。

```
WSAStartup(…);
    ……
    HOSTENT *host = gethostbyname("www.seu.edu.cn");
    struct in_addr addr;
    if(host != NULL){
        for(int i = 0; host->h_addr_list[i] != NULL; i++){
            memset(&addr, 0, sizeof(addr));
            memcpy(&addr.S_un.S_addr,
                    host->h_addr_list[i],
                    host->h_length);
            printf("%s: %s\n",host->h_name, inet_ntoa(addr));
        }
```

```
    }
        ……
        WSACleanup();
```

输出结果如下：

```
seic22.seu.edu.cn: 202.119.24.32
```

17.3　端　　口

从 17.2.1 节 INET 地址结构的介绍中可以发现，要惟一确定一个 INET 地址，除了知道 IP 地址外，还需要知道 16 位的端口号。事实上，当数据从网络底层传来，并逐层解开至传输层时，端口号是系统了解应该将数据交付哪个应用程序处理的依据。不同的服务对应于不同的端口号，这些端口号通常被划分为以下几段。

- 0，不使用，如果你的应用程序选择了 0 为端口，那么系统将为它随机分配一个 1024～5000 之间的值。
- 1～1023，知名端口，限定为知名服务使用，如 ftp 的 21 端口、http 的 80 端口等。这些知名端口号由 IANA 组织统一管理。
- 1024～5000，可以被任意的客户端程序使用。客户端通常对它所使用的端口号不关心，只需保证其惟一性即可，因此又称作临时端口号（即这种联系的存在很短暂）。大多数 TCP/I P 实现给临时端口分配 1024～5000 之间的端口号。
- 5001～65535，为其他服务器程序预留，这些服务是 Internet 上不常用的。如果需要开发自己的服务器程序，就应该使用 5000 以外的端口号。

17.4　字节序问题

虽然第 3 章讲述了字节序的问题，这里需要从实际编程的角度重申这个问题的重要性。不同的计算机系统采用不同的字节序存储数据，这为网络应用程序之间数据的交互造成了很大的麻烦。同样一个两字节的 16 位整数，在内存中存储的方法不同：一种是将低字节存储在起始地址，称为“Little-Endian”字节序，由 Intel 等体系结构采用；另一种是将高字节存储在起始地址，称为 “Big-Endian”字节序，由 Macintosh 等体系结构采用。以 16 位整数 0x0102 为例，两种方式下内存布局情况如图 17.2 所示。

通常把某给定系统所用的字节序称为主机字节序，而把网络标准的“Big-Endian”称为网络字节序。由于两种字节序都广为使用，这就给不同类别主机之间的网络数据交互设计带来了一定的麻烦。为了解决这个问题，在套接口 API 中提供了一组字节序处理函数——htonx、ntohx，其中 h 表示 host，n 表示 network，x 或者是 s（short）或者是 l（long）。作为编程人员，只需要简单地调用这些函数，而不用考虑本地的字节序与网络序是否有差别。

1．htons 函数将一个无符号的短整型数（16 位）转化为“Big-Endian”的网络字节序。定义如下：

```
u_short htons(u_short hostshort);
```

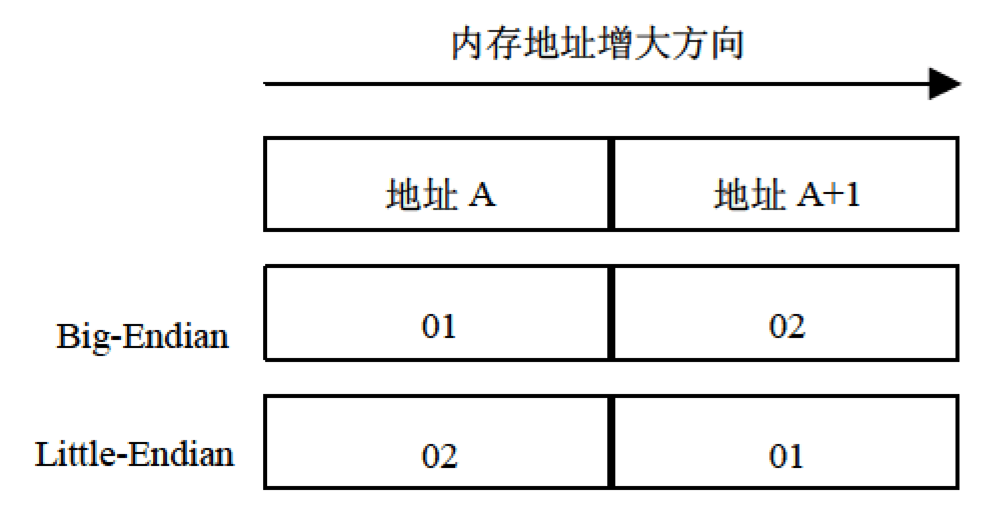


图 17.2　字节序问题

- hostshort：[IN]，16 位的主机序短整型数。
- 返回值：变换为网络序的值。

2．ntohs 函数操作与 htons 恰恰相反，它将一个网络序的 16 位无符号整数转化为主机序。定义如下：

```
u_short ntohs(u_short netshort);
```

- netshort：[IN]，16 位的网络序短整型数。
- 返回值：返回主机序的值。

3．htonl、ntohl 函数分别与 htons 和 ntohs 相同，差别仅在于这两个函数操作的是 32 位的无符号长整型数。

17.5　三种套接口类型和两种 I/O 模式

17.5.1　套接口的类型

套接口是网络通信端点的一种抽象概念，它向应用程序提供数据发送和接收的功能。进行网络编程时经常会碰到三种基本的套接口类型：SOCK_STREAM、SOCK_DGRAM 和 SOCK_RAW。

- SOCK_STREAM：流套接口，对应于 TCP 协议，向应用程序提供可靠的面向连接的数据传输服务。也称面向连接的套接口、TCP 套接口等。
- SOCK_DGRAM：数据报套接口，对应于 UDP 协议，向应用程序提供不可靠的、非连接的数据报通信方式。也称无连接套接口、面向消息套接口、UDP 套接口等。
- SOCK_RAW：原始套接口，可以读写 ICMP、IGMP 报文，可用于从 IP 头起构造自己的报文。

17.5.2　I/O 模式

套接口 I/O 模式是指套接口进行输入、输出时调用的那些函数操作的工作模式。Winsock 支持两种 I/O 模式：阻塞（BLOCK）和非阻塞（NONBLOCK）。

在阻塞模式下，I/O 操作完成前，执行该操作的 Winsock 函数（比如 send/sendto 和 recv/recvfrom）不会立即返回，它会一直等待下去直到所需进行的操作完成为止。

在非阻塞模式下，Winsock 函数无论操作是否已完成，都会立即返回。通常会发现这些函数操作失败，并且会得到 WSAEWOULDBLOCK 的错误码。它意味着所进行的函数操作在函数调用的这段时间内没有完成，必须重新进行尝试。

阻塞模式与非阻塞模式相比较，从编程角度来说前者更便于使用，但从程序运行的效率来说，由于阻塞调用会使得所在的线程（如果是主线程那么就是整个程序）等待在该 I/O 操作上，因此后者的效率更高。

默认情况下，Winsock 函数都以阻塞模式进行工作。这些函数主要涉及到连接（connect）、接受连接（accept）、发送数据、接收数据、关闭套接口等。可以使用 ioctlsocket 函数来改变套接口的 I/O 模式，其函数定义如下：

```
int ioctlsocket(SOCKET s, long cmd, u_long FAR *argp);
```

其中 s 是待处理的套接口描述字（简称套接字，与套接口概念有所不同），cmd 是对该套接口进行的操作（模式修改），argp 是 cmd 的参数。其中 cmd 可以是 FIONBIO、FIONREAD 或者 SIOCATMARK，这里只介绍 FIONBIO，该命令用于开启/禁止套接口的非阻塞 I/O 模式。在下面两段代码中，套接口 s 分别工作在非阻塞模式和阻塞模式下。

```
// 代码段1：使用非阻塞模式
u_long bNonblock = 1;
    if(ioctlsocket(sock, FIONBIO, &bNonblock) == SOCKET_ERROR){
    HandleError("ioctlsocket");
    }
// 代码段2：禁止非阻塞模式，即使用阻塞模式
u_long bNonblock = 0;
    if(ioctlsocket(sock, FIONBIO, &bNonblock) == SOCKET_ERROR){
    HandleError("ioctlsocket");
    }
```

但是，由于非阻塞调用会频繁返回 WSAEWOULDBLOCK 错误，所以在任何时候，都应仔细检查返回的错误代码，并做好调用失败的准备。如果每次碰到 WSAEWOULDBLOCK 错误时，只是简单地等待然后再次调用该 I/O 函数，那么这种轮询工作方式的效率甚至会低于阻塞 I/O 方式。如何充分利用非阻塞 I/O 带来的效率提高，将在 18 章 Winsock 的 I/O 模型中进行介绍。

17.6　基本套接口函数

C/S 结构是网络应用系统的常见模型，Sockets 接口可以很好地满足客户端和服务器之间进行通信的需求。以开发一个基于 TCP 的服务器程序为例，通常的流程是这样的：首先初始化 Winsock，创建一个 Socket，绑定并监听本地的某个特定端口，accept 接受客户端的连接，在 accept 操作返回的 Socket 上进行数据通信，关闭 Socket，最后结束 Winsock 的使用。下面



大致按照这个顺序介绍需要调用的函数。

17.6.1 WSAStartup

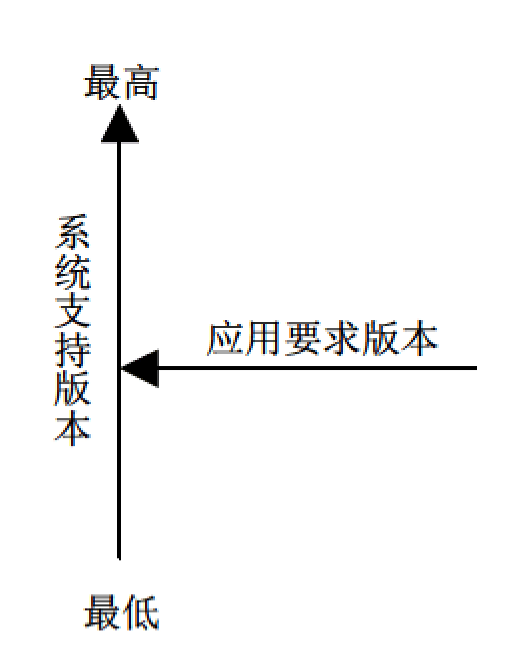
与伯克利套接口 API 不同，在使用 Winsock 库函数之前，必须先调用函数 WSAStartup，该函数负责初始化动态连接库 Ws2_32.dll。函数定义如下：

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

- wVersionRequested：[IN]，一个 WORD（双字节）型数值，指定了应用程序需要使用的 Winsock 规范的最高版本。其中主版本号在低字节，次版本号在高字节。如果对版本并不关心，那么可以直接将常量值 WINSOCK_VERSION 赋给 wVersionRequested，该常量在 Winsock2.h 中定义，表示当前的 Winsock 版本。假设希望的版本号是 1.2（演示，实际不存在），那么可以使用如下代码：wVersionRequested = 0x0201，或者 wVersionRequested = MAKEWORD（1, 2），其中 MAKEWORD 是一个宏定义，可将两个字节组装成一个 WORD。WORD MAKEWORD(BYTE blow, BYTE bHigh)换成版本号就应该是 MAKEWORD(BYTE bMainVersion, BYTE bLowVersion)。
- lpWSAData：[OUT]，指向 WSADATA 数据结构的指针，该结构用于返回本机的 Winsock 系统实现的信息。经常使用的是该结构的 WhighVersion 和 wVersion 两个域，前者表示系统支持的最高版本，后者是系统希望调用者使用的版本。
- 返回值：如果函数操作成功，返回 0；否则返回错误码。需要注意的是，由于 ws2_32.dll 尚未初始化，此时无法调用函数 WSAGetLastError。WSAGetLastError 是 Winsock 提供的辅助函数，当调用 Winsock 函数出错时，可以使用该函数查出系统的出错代码，其函数定义如下：

```
int WSAGetLastError (void);
```

- 注释：WSAStartup 是任何使用 Winsock 的应用程序或者 DLL 首先必须调用的 Winsock 库函数。一方面它完成初始化 ws2_32.dll 的工作，另一方面它也可用于在应用程序（或 DLL）与系统 Winsock 库之间进行版本协商。只有当应用要求的版本（希望使用的 Winsock 的最高版本）等于或者高于系统支持的最低版本（下限），那么该函数操作成功并且在 WSADATA.WhighVersion 中返回系统支持的最高版本，在 WSADATA.wVersion 中返回系统支持的最高版（上限）和 wVersionRequested 之间的较小值，也就是系统希望使用的版本号。此时应该判断该版本是否合适。



- 示例，见如下代码：

```
*************************** 程序17.1 WSAStartup ***************************
#pragma comment(lib, "ws2_32.lib")
```

```
#include <STDIO.H>
#include <WINSOCK2.H>

int main(int argc, char* argv[])
{
    WSAData wsaData;
    WORD wVersionReq = MAKEWORD(0, 1);
    int ret = WSAStartup(wVersionReq, &wsaData);
    if(ret != 0){
        printf("%d\n", ret);
        return -1;
    }
    else{
        printf("High: %x Use: %x\n", wsaData.wHighVersion, wsaData.wVersion);
        WSACleanup();
        return 0;
     }
}
***********************************************************************
```

其中一些代码，如第一行#pragma 的使用，读者可以暂时不予理会，我们将在本章的最后一节给出一个完整的例子并进行详细的分析。表 17.1 给出了在指定不同的输入参数 wVersionReq 时程序的输出结果。

表17.1　版本协商输出结果

wVersionReq	描　述	结　果	解　释
MAKEWORD(0,1)	希望使用 0.1 版本	Error: 10092	0.1 低于系统的最低版本，出错
0x0002	希望使用 2.0 版本	High: 202 Use: 2	系统最高支持 2.2 版本，建议使用 2.0 版本
MAKEWORD(3,1)	希望使用 3.1 版本	High: 202 Use: 202	系统最高支持 2.2 版本，建议使用 2.2 版本
WINSOCK_VERSION		High: 202 Use: 202	系统最高支持 2.2 版本，建议使用 2.2 版本

17.6.2　socket

在初始化 Winsock 的使用后，就可以调用 socket 函数来创建套接口了，函数定义如下：

```
SOCKET socket (int af, int type, int protocol);
```

- af：[IN]，指定协议族，一般都为 AF_INET，对应于 internet 协议。
- type：[IN]，指定套接口类型，常用的有 3 种，如表 17.2 所示。
- protocol：[IN]，指定所用的协议。
- 返回值：如果没有错误发生，函数返回一个新的套接口的描述字；否则返回常量值 INVALID_SOCKET，此时可以调用 WSAGetLastError 来查出系统的错误代码。



- 注释：SOCKET 是 Winsock 定义的套接口类型，但事实上，与 Berkeley 标准 API 一样，它本质上是一个整型数。在 Winsock2.h 中，SOCKET 被定义为：

```
typedef u_int SOCKET;
```

表17.2　套接口类型

SOCK_STREAM	提供可靠的、双向的、基于连接的字节流传输服务，对应于 TCP 协议
SOCK_DGRAM	提供不可靠的、非连接的数据报服务，通常这种数据报大小固定并且较小，对应于 UDP 协议
SOCK_RAW	可以读写 ICMP、IGMP 分组，可以读写内核不处理的包含特殊 IP 协议段的 IP 报文，利用原始套接口加上 IP_HDRINCL 选项可以构造自己的 IP 报头

在创建 SOCKET 时，对于 TCP、UDP 类型，socket 函数的第三个参数通常可以设置为 0（这是因为根据 AF_INET + SOCK_STREAM/SOCK_DGRAM，即可惟一确定所使用的协议）；只有需要创建原始套接口，才需要设置 protocol 参数。

- 示例：下面 3 段代码分别创建 TCP、UDP 和原始套接口（ICMP 协议）。

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
SOCKET sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

17.6.3　bind

bind 函数将一个本地的传输层地址与已创建的套接口联系起来。一般来说，作为客户端程序，不用关心它的本地地址是什么，也就没有必要调用 bind 函数，系统会在通信之前（对于 TCP，通常是在客户端 connect 或者服务器端 listen；对于 UDP 是在 sendto）自动为它选择一个本地地址（端口一般为 1024～5000，端口的选择见 17.3 节）加以绑定。服务进程则必须要绑定到一个为客户端所知的地址上，所以在接受连接或接收数据报之前必须调用 bind。无论是显式地调用 bind 函数，还是由系统隐式地进行地址绑定，我们都将此时的套接口称为已绑定套接口。函数定义如下：

```
int bind(SOCKET s, const struct sockaddr FAR *name, int namelen);
```

- s：[IN] 未绑定的套接口描述字。
- name：[IN]，指向供套接口使用的本地地址的通用地址指针。
- namelen：[IN]，name 参数的长度。
- 返回值：如果没有错误发生，函数返回 0；否则返回 SOCKET_ERROR，可以调用 WSAGetLastError 来获取具体的错误代码。一个常见的错误是 WSAEADDRINUSE，将在稍后介绍服务器设计时讨论这个问题。
- 注释：该函数尤其需要注意的是第二个参数。在填充地址信息时，通常都使用 INET 地址结构，因此在调用函数时必须进行强制转换。
- 对于多穴主机，绑定地址时，或者选择其中一个接口或者选择任意接口（INADDR_ANY），不能选择绑定其中的若干接口。表 17.3 显示了在一个多网络接

口主机环境下，为套接口指定不同的地址数据的结果。

表17.3　端口和地址的绑定

IP	端　口	结　果
INADDR_ANY	0	内核选择某适配器的地址和 1024～5000 之间的端口
INADDR_ANY	非 0	内核选择某适配器的地址，进程指定端口
本地一适配器地址	0	进程指定 IP 地址，内核选择 1024～5000 之间的端口
本地一适配器地址	非 0	进程指定 IP 地址和端口

- 示例：下面的代码创建了一个流套接口，并绑定至本地 IP-202.119.9.199 的 9999 端口。

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in local;
memset(&local, 0, sizeof(local));
local.sin_addr.s_addr = inet_addr("202.119.9.199");
local.sin_family = AF_INET;
local.sin_port = htons(9999);
if(bind(sock, (struct sockaddr *) &local, sizeof(local)) == SOCKET_ERROR){
    printf("Error: %d\n", WSAGetLastError());
    WSACleanup();
    return -1;
}
```

17.6.4　listen

该函数仅被 TCP 服务器端使用，负责通知协议内核用户进程准备接收套接口上的连接请求，它同时也指定了在该套接口上可以排队等待的连接数的门限值。函数定义如下：

```
int listen(SOCKET s, int backlog);
```

- s：[IN]，已绑定但尚未连接的套接口描述字。
- backlog：[IN]，待处理的连接队列的最大长度。如果设置为常量值 SOMAXCONN，底层的网络服务驱动会自动为该套接口设置最大的队列长度值。
- 返回值：如果没有错误发生，函数返回 0；否则返回 SOCKET_ERROR，同样可以调用 WSAGetLastError 以获取具体的错误码。
- 注释：TCP 套接口有主动套接口和被动套接口之分，当调用 socket 函数创建一个流 SOCKET 时，系统会假设该套接口为主动，这时可以调用 connect 函数以主动发起连接，这就是客户端 SOCKET；如果调用的不是 connect 而是 listen 函数，那么系统将该未连接 SOCKET 改为被动，并接受指向它的连接请求，这就是服务器端 SOCKET，也称监听套接口。

系统为每一个监听套接口维护一个待处理连接的队列，该队列由两个子队列组成：未完成连接队列和已完成连接队列，区分的标准是是否已完成 TCP 三步握手，如图 17.3 所示。

客户端 connect，
进行三步握手
服务器端 accept
未完成连接队列
已完成连接队列
三步握手中
三步握手完毕

图 17.3　连接队列

因此，有参数 backlog＝ 待处理连接队列长度 ＝ 未连接队列长度 ＋ 已连接队列长度。

❒ 示例：用一个小例子来演示如何创建服务器端的监听套接口，并根据这个例子来检测在 Windows 系统下 backlog 值的意义。该程序的设计非常简单，取不同的BACKLOG 常量值为流套接口调用 listen 函数，然后 Sleep 足够长时间，使服务器无法处理进入的连接请求；同时客户端发起多次 connect 连接，根据返回值来判断服务器的连接队列的状态。

```
***************************** 程序17.2 Listen ***************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>

#define BACKLOG     1024

int main(int argc, char* argv[])
{
    WSAData wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);

    SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in local;
    memset(&local, 0, sizeof(local));
    local.sin_addr.s_addr = INADDR_ANY;
    local.sin_family = AF_INET;
    local.sin_port = htons(9999);
    bind(sock, (struct sockaddr *) &local, sizeof(local));

    if(listen(sock, BACKLOG) == SOCKET_ERROR){
        printf("Listen error %d!\n", WSAGetLastError());
    }

    Sleep(1000000);
    closesocket(sock);
```

```
    WSACleanup();
    return 0;
}
**********************************************************************
```

为 BACKLOG 常量设置不同的值，有如下结果：

① BACKLOG = 0，当第 2 个连接到来时，服务器直接返回 TCP-RST 报文，客户端 connect 返回 10061 错误（WSAECONNREFUSED - Connection refused）。

② BACKLOG = 1，当第 2 个连接到来时，服务器直接返回 TCP-RST 报文，客户端 connect 返回 10061 错误。

③ BACKLOG = 2，当第 3 个连接到来时，服务器直接返回 TCP-RST 报文，客户端 connect 返回 10061 错误。

④ BACKLOG = 6，当第 7 个连接到来时，服务器直接返回 TCP-RST 报文，客户端 connect 返回 10061 错误。

⑤ BACKLOG = 1024，当第 202 个连接到来时，服务器直接返回 TCP-RST 报文，客户端 connect 返回 10061 错误。

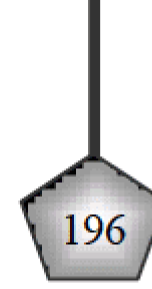
由此可以得出结论，假设系统的待处理连接队列长度为 X，当 backlog 不在 1～X 范围内时，系统会自动将其改为临近的可接受的值，即如果 backlog<1，则改为 1，如果 backlog>X，则改为 X；当连接队列已满，并有新的客户连接进来时，Win2000 系统会直接返回 TCP-RST 报文，因此客户端 connect 会直接收到错误反馈。这种处理方式与 Unix 系统不同，事实上，忽略后续 SYN 报文还是返回 TCP-RST 都是 POSIX 标准所允许的，历史上，所有源于 Berkeley 的 UNIX 实现都是忽略新的 SYN 请求。

17.6.5　accept

TCP 服务器套接口在调用了 listen 之后，还应该调用 accept 来等待接受连接请求。accept 函数定义如下：

```
SOCKET accept(SOCKET s, struct sockaddr FAR *addr, int FAR *addrlen);
```

- s：[IN]，处于监听状态的套接口描述字。
- addr：[OUT]，用于接收外来连接的地址信息。可选。如果暂时不关心该地址信息，那么可以输入 NULL。
- addrlen：[INOUT]，这是第一次碰到 INOUT 类型的参数，也可以称为值-结果类型参数。这种参数一方面接收调用者给出的输入值，一方面用于返回结果。在调用 accept 函数前，应该将 addrlen 设定为由 addr 所指的 INET 地址结构的长度；调用完毕后，addrlen 会返回内核保存该地址所用存储空间的精确字节数。如果输入的 addrlen 值小于内核所需的存储空间大小，那么函数调用将返回错误 10014（WSAEFAULT - Bad address）。可选。
- 返回值：如果没有错误发生，返回一个新的已连接套接口的描述字；否则返回 INVALID_SOCKET。
- 注释：如果没有错误发生，accept 函数将返回一个新的已连接的套接口描述字（称



之为服务套接口)，该套接口绑定了本地的网络地址与监听端口；同时，原有的监听套接口将仍然处于监听状态，新的连接请求可以通过再次的 accept 调用而获得接受。一般情况下，服务器会同时处理很多并发连接，那么也就存在着很多服务套接口，那么应用程序如何知道发往本地的同一端口的 TCP 数据该由哪个服务套接口来处理呢？事实上，系统内核解决了该问题，每个 TCP 连接都由<源 IP，源端口>—<目的 IP，目的端口>惟一确定，内核会根据 TCP 数据的源地址/目的地址，自动将数据放入与该地址相对应的套接口的数据缓冲区中。

当调用 accept 时输入的第二、三个参数都为空指针时，函数将不返回连接发起方的地址，如果此后（必须在断开连接之前）需要了解该信息，那么可以使用函数：

```
int getpeername(SOCKET s, struct sockaddr FAR *name, int FAR *namelen);
```

其中 s 是已连接的套接口的描述字（accept 的返回值），namelen 是 INOUT 类型参数。与该函数相对应，如果想知道本地某个已绑定套接口的地址信息，那么可以使用函数：

```
int getsockname(SOCKET s, struct sockaddr FAR *name, int FAR *namelen);
```

需要注意的是如果该套接口尚未绑定，或者绑定了 INADDR_ANY 地址但尚未调用 connect 函数，那么函数将返回 10022 错误（WSAEINVAL）。

- ❒ 示例：下面的代码完成了接受连接并输出对方地址的功能，其中 sock 是 TCP 监听套接口。

```
struct sockaddr_in from;
int len = sizeof(from);
if(accept(sock, (struct sockaddr *) &from, &len) == SOCKET_ERROR){
    printf("accept :%d", WSAGetLastError());
}
else{
    printf("%s\n", inet_ntoa(from.sin_addr));
}
```

17.6.6 connect

在 17.6.4 节中提到，对于流套接口有主动和被动之分。在客户端，当创建了一个主动套接口后，就可以使用 connect 函数来连接服务器。但是，这并不是说只有面向连接的套接口才能调用 connect 函数，在下文将对这个问题进行详细的分析。函数定义如下：

```
int connect(SOCKET s, const struct sockaddr FAR *name, int namelen);
```

- ❒ s：[IN]，未绑定的套接口描述字。
- ❒ name：[IN]，指向目标地址的指针，目标地址中必须包含 IP 和端口信息。
- ❒ namelen：[IN]，name 的长度。
- ❒ 返回值：如果没有错误发生，返回 0；否则返回 SOCKET_ERROR。
- ❒ 注释：connect 函数既可用于面向连接套接口，也可用于无连接套接口，两种情况下的作用是不同的。

✧ 无连接套接口：对于无连接的套接口（例如 SOCK_DGRAM），connect 函数仅仅起到在该套接口与目标地址之间建立默认的对应关系的作用，没有任何网络数据（协议）的交互发生。在 connect 之后，套接口变为已绑定和已连接状态，这时可以直接使用 send，而不是用 sendto 来向该地址发送数据；同时，内核会丢弃所有发送给该套接口的源地址不是 connect 地址的报文。如果要变更这种关系，可以再次调用 connect 函数：如果此时 name 和 namelen 两个参数均为空指针，那么就会将该套接口恢复为未连接状态，此时再调用 send 函数，系统会提示 WSAENOTCONN 错误码。

✧ 面向连接套接口：对于面向连接的套接口（SOCK_STREAM），函数 connect 会引起调用端主动进行 TCP 的三次握手过程。其结果通常是成功连接、WSAETIMEDOUT（多次发送 SYN 报文，始终未收到回复）、WSAECONNREFUSED（目标主机返回 TCP-RST）等。从图 17.4 中可以看到系统为一次成功的 connect 所作的网络交互（连接 202.119.24.32 的 80 端口）。

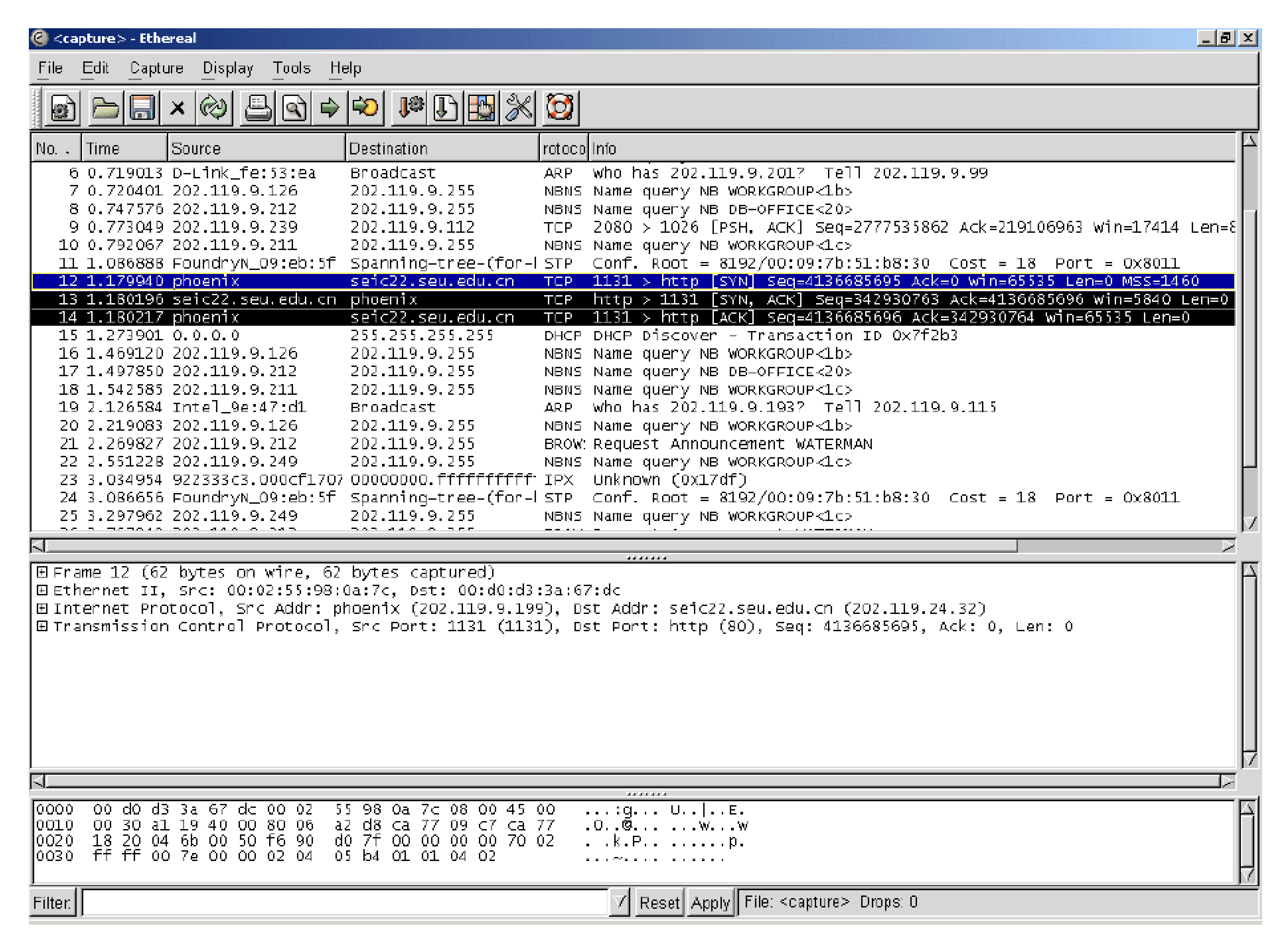


图 17.4　connect 引起的网络数据交互

17.6.7　recv 和 send

recv 函数从套接口接收数据，该套接口可以是面向连接的，也可以是无连接的。如果是面向连接的套接口，那么它必须是已连接的；如果是无连接套接口，那么它必须是已绑定的。函数定义如下：

```
int recv(SOCKET s, char FAR *buf, int len, int flags);
```

- ❒ s：[IN]，套接口描述字。
- ❒ buf：[OUT]，用于接收数据的缓冲区指针。
- ❒ len：[IN]，应用程序提供的缓冲区大小。
- ❒ flags：[IN]，设定 recv 方式的标志值，MSG_PEEK 或者 MSG_OOB。前者用于把输入队列中的数据读至用户缓冲区，但是并不相应地将该数据从输入队列中删除；后者用于读取带外数据。通常把该参数置为 0。
- ❒ 返回值：如果没有错误发生，recv 返回接收的字节数；否则返回 SOCKET_ERROR。需要注意的是，返回结果 0 并不表明有错误发生：如果是 UDP 套接口，那么说明读到了一个没有应用数据的纯 UDP 头报文；如果是 TCP 连接，这就意味着对方已关闭了连接。
- ❒ 注释：对于 recv 函数的使用，在面向连接和非连接的套接口中是不同的。通常情况下，recv 函数应用于流套接口。

✧ 面向连接（SOCK_STREAM）

调用 recv 之前，套接口必须是已连接的；recv 函数只接收来自已连接的远端地址的数据。每次 recv 调用都将能读取当前可获得的所有数据，如果可读取数据量比用户缓冲区大，那么 recv 操作将先读取前面部分的数据，因此无法保证一次 recv 操作就能将所需要的数据全部读到，所以可以将 recv 编码在一个循环中，直到读取了所有需要的数据，或者 recv 返回为 0（表明远程端口关闭连接）或 SOCKET_ERROR（错误发生）时才中止循环。

✧ 无连接（如 SOCK_DGRAM）

调用 recv 之前，套接口必须是已绑定的。如果该套接口调用了 connect 函数，那么只有来自 connect 函数中指定的远端地址的报文才会被接受。如果 recv 操作时，系统输入队列中的待读取数据报比用户缓冲区大，那么数据报的前面部分的数据将被读至用户缓冲区，超出部分将被丢弃，同时 recv 操作出错，返回错误码 10040（WSAEMSGSIZE - Message too long）。因此，在接收 UDP 报文时，如果 recv 返回 SOCKET_ERROR，还应该检查具体的错误码，WSAEMSGSIZE 或许是应用程序可以接受的。

与 recv 函数相对应，send 函数从一个已连接套接口发送数据。函数定义如下：

```
int send(SOCKET s, const char FAR *buf, int len, int flags);
```

- ❒ s：[IN]，已连接套接口的描述字，可以是面向连接的，也可以是无连接的。
- ❒ buf：[IN]，待发送数据的缓冲区指针。
- ❒ len：[IN]，待发送数据的字节数。
- ❒ flags：[IN]，操作标志，可置为 0。
- ❒ 返回值：如果没有错误发生，send 返回成功发送的字节数，对于非阻塞套接口来说，该值可能小于 len；否则，返回 SOCKET_ERROR。
- ❒ 注释：该函数必须用于已连接套接口。对于数据报套接口，应注意待发送的报文大小的上限问题，如果 send 的报文过大，将返回 WSAEMSGSIZE 错误码。此外，成功的 send 操作并不保证数据被成功地传递给了目标主机，这个问题将在 17.6.8 节做深入地阐述。一般情况下，send 函数应用于流套接口。

17.6.8　recvfrom 和 sendto

recvfrom/sendto 函数类似于 recv/send，一般情况下，前者应用于数据报套接口，后者用于流套接口。尽管没有理由将 recvfrom/sendto 应用于 TCP，但确实也可以这么做。

recvfrom 的函数定义如下：

```
int recvfrom(SOCKET s, char FAR* buf, int len, int flags,
struct sockaddr FAR *from, int FAR *fromlen
);
```

- 参数 s、buf、len 和 flags 与 recv 完全相同。
- from：[OUT]，用于保存接收到的数据的源地址的缓冲区指针。可选。
- fromlen：[INOUT]，from、fromlen 两个参数类似于 accept 函数的后两个参数。在调用 recvfrom 函数之前，应该将由 fromlen 置为由 from 所指的 INET 地址结构的长度；调用完毕后，fomrlen 返回内核保存该地址的所用存储空间的精确字节数。如果输入的 fromlen 值小于内核所需的存储空间大小，那么函数调用将返回错误 10014（WSAEFAULT - Bad address）。可选。
- 返回值：如果没有错误发生，recvfrom 返回接收的字节数；否则返回 SOCKET_ERROR。需要注意的是，返回结果 0 并不表明有错误发生，通常来说这是 UDP 套接口读到了一个纯 UDP 报文头。
- 注释：recvfrom 与 recv 函数的差别仅仅是前者在获取数据的同时还可以获得该数据的源地址。如果将 recvfrom 的后两个参数设置为空指针，那么它就等价于 recv 函数。对于数据报套接口来说，每个 UDP 套接口都有一个输入缓冲区，到达的报文都进入该缓冲区保存，当进程调用 recvfrom，那么缓冲区中的报文就会以 FIFO 的顺序返回给进程。

sendto 函数定义如下：

```
int sendto(SOCKET s, const char FAR *buf, int len, int flags,
           const struct sockaddr FAR *to, int tolen
);
```

- 参数 s、buf、len 和 flags 与 send 函数参数一致。
- to：[IN]，指向目标套接口地址的指针。如果套接口是已连接的，那么该指针可为空。可选。
- tolen：[IN]，地址结构 to 的大小。
- 返回值：如果没有错误发生，sendto 返回成功发送的数据的字节数，该值可能比 len 要小；否则返回 SOCKET_ERROR。
- 注释：sendto 函数通常用于无连接套接口（如 UDP），向指定目标地址发送数据报；即使该套接口已经调用了 connect 函数，传送该报文的目标地址仍然由 to 参数决定。对于一个面向连接的套接口，参数 to 和 tolen 将被忽略，sendto 等价于 send。
- 示例：对于 UDP 协议来说，大家都知道成功地调用 sendto 并不意味着数据确实被发送到了目标地址，然而，事实上甚至不能据此推断数据确实从本地网络接口发送

出去了。

下面的代码向同一局域网中的一台离线主机发送 UDP 报文，根据网络监视器截获的报文可发现：由于该主机并不在线，本机发送的 ARP 请求得不到响应，因此 sendto 发送的数据报无法组装成链路层的数据帧，也就无法被发送出本地网络接口。然而此时的 sendto 函数依然给出了正常的返回值。

得出的结论是：成功的 sendto 调用不能确保数据被发送出，更不能保证成功地到达目标主机。如下：

```
**************************** 程序17.3 UDPSendto****************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>

#define OFFLINE_HOST    "202.119.9.4"
#define DATANUM         6

void HandleError(char*);

int main(int argc, char* argv[])
{
    WSAData wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);
    SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in to;
    memset(&to, 0, sizeof(to));
    to.sin_addr.S_un.S_addr = inet_addr(OFFLINE_HOST);
    to.sin_family = AF_INET;
    to.sin_port = htons(9999);

    char *buf = "Hello!";
    int res = sendto(sock, buf, DATANUM, 0, (struct sockaddr *) &to, sizeof(to));
    if(res == SOCKET_ERROR){
        HandleError("sendto");
    }
    else
        printf("Send out %d bytes!\n", res);

    closesocket(sock);
    WSACleanup();
    return 0;
}

void HandleError(char *func)
{
    int errCode = WSAGetLastError();
```

```
    char info[65] = {0};
    _snprintf(info, 64, "%s:        %d\n", func, errCode);
    printf(info);
}
**********************************************************************
```

输出结果是“Send out 6 bytes!”，但事实上嗅包器告诉我们除了没有回应的 ARP 请求“Who has 202.119.9.4? Tell 202.119.9.199”（202.119.9.199 是发送端）外，没有任何的 UDP 报文发出，如图 17.5 所示。

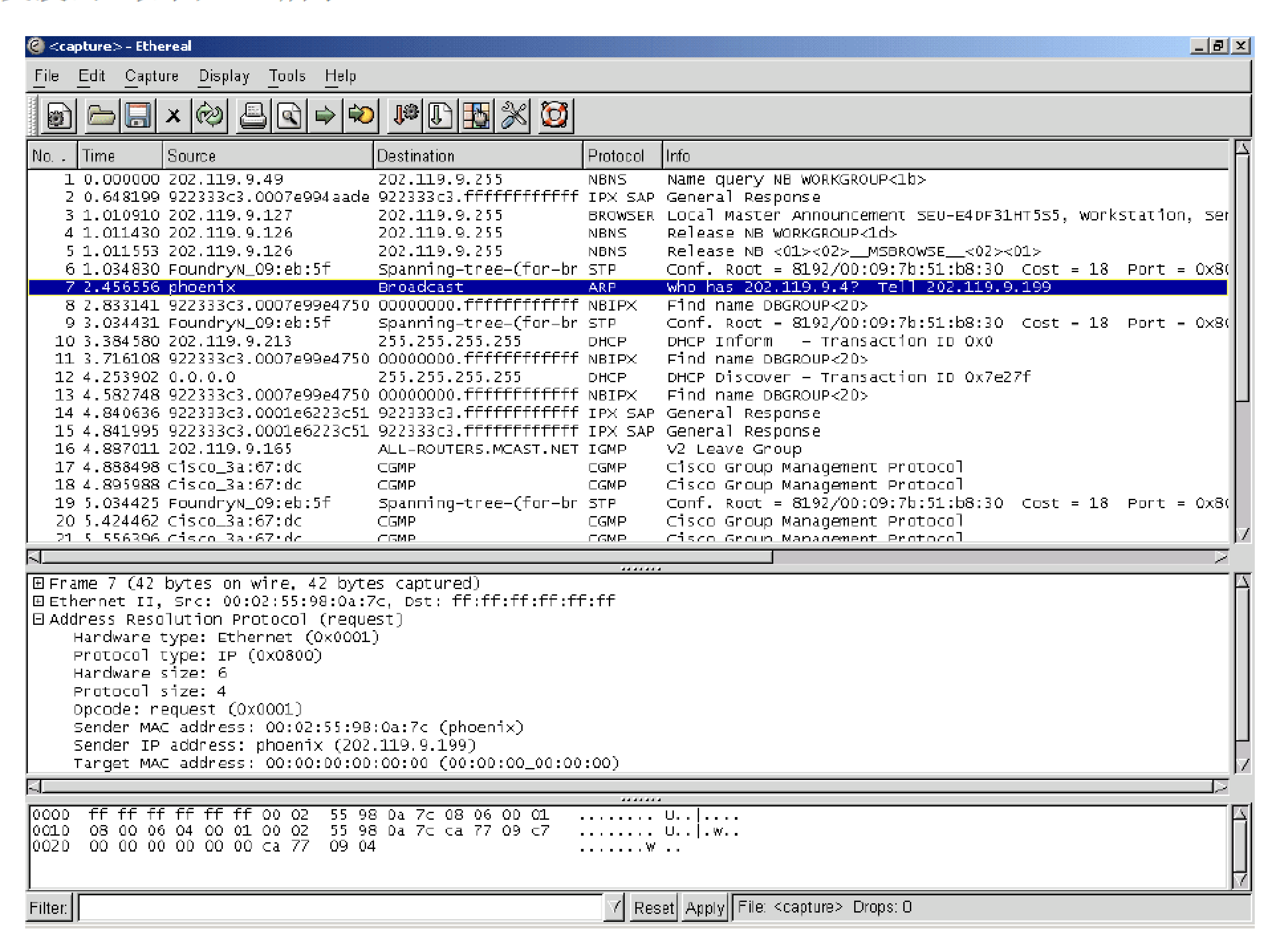


图 17.5　sendto 的问题

发送一个 0 字节的数据报也是可行的，它将使得一个不带任何应用层数据的 UDP 报文头被发送出去。把上例中 OFFLINE_HOST 改为实际上在线的主机 202.119.9.242，把 DATANUM 改为 0，并在 202.119.9.242 上运行下面的简单接收程序：

```
SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);

struct sockaddr_in addr;
int len = sizeof(addr);
memset(&addr, 0, sizeof(addr));
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_family = AF_INET;
addr.sin_port = htons(9999);
```

```
if(bind(sock, (struct sockaddr *) &addr, len) == SOCKET_ERROR){
    printf("bind: %d", WSAGetLastError());
}

memset(&addr, 0, sizeof(addr));
char buf[120] ={0};
ret = recvfrom(sock, buf, 120, 0, (struct sockaddr *) &addr, &len);
if( ret == SOCKET_ERROR){
    printf("recv: %d\n", WSAGetLastError());
}
else
    printf("recv %d bytes from %s!\n", ret, inet_ntoa(addr.sin_addr));
```

接收端的运行结果为“recv 0 bytes from 202.119.9.199!”，因此可以得出结论：发送 0 字节的数据报是可以的；在接收端，recvfrom 返回 0 也是正常的现象，并不说明对方已关闭来连接，这与 TCP 套接口上 recv 返回 0 不同。如图 17.6 所示。

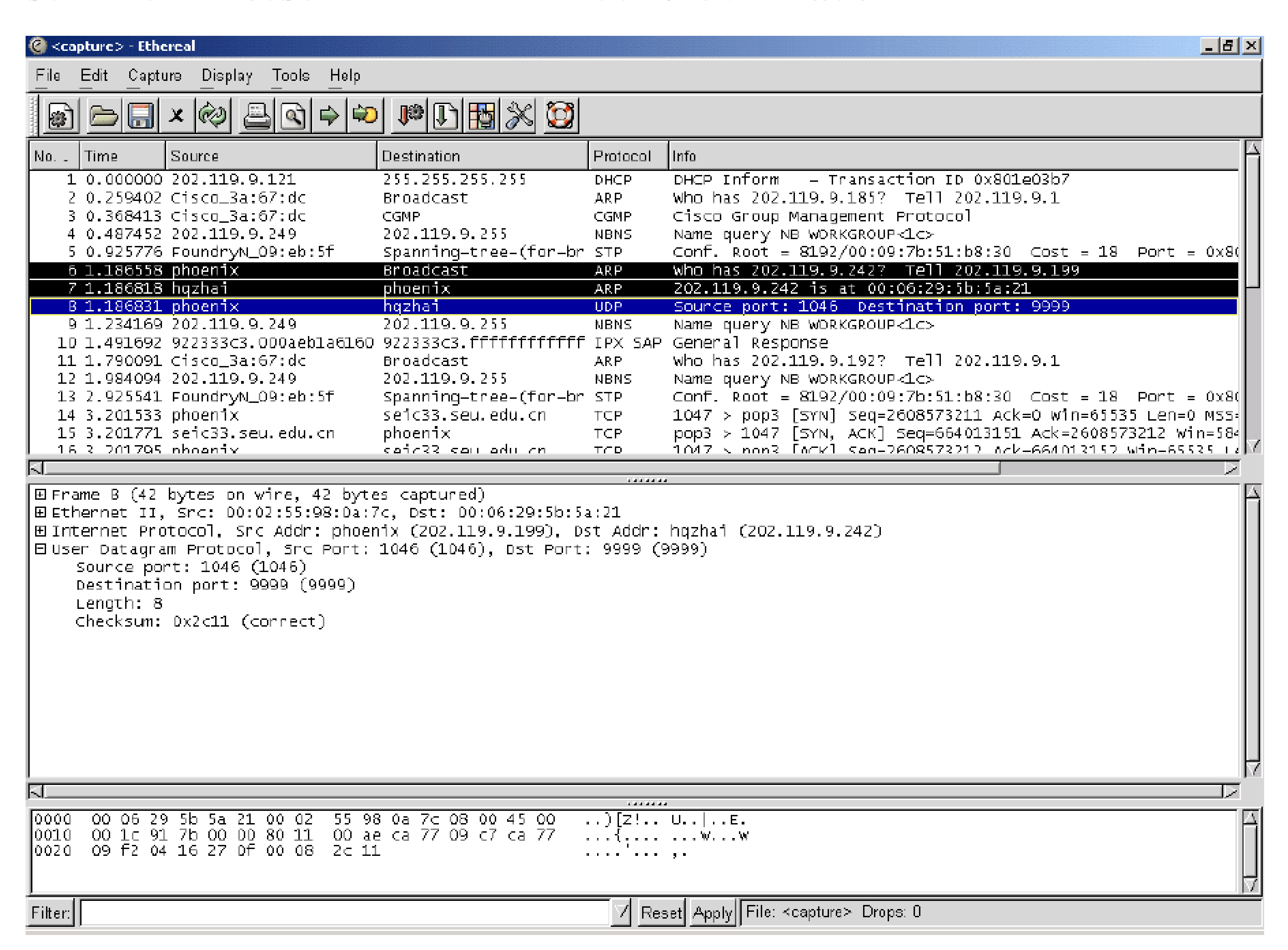


图 17.6　发送 0 字节 UDP 报文

17.6.9　closesocket

该函数关闭一个存在的套接口。函数定义如下：

```
int closesocket(SOCKET s);
```

- s：[IN]，待关闭的套接口描述字。

- 返回值：如果没有错误发生，返回 0；否则 SOCKET_ERROR。
- 注释：该函数用于关闭套接口并释放套接口描述字 s，调用该函数后，任意对 s 的操作都将引起 WSAENOTSOCK 的发生。

17.6.10　WSACleanup

在完成了所有的套接口通信及函数应用后，就可以调用 WSACleanup 来终止 ws2_32.dll 的使用了。函数定义如下：

```
int  WSACleanup (void);
```

- 返回值：如果没有错误发生，返回 0；否则 SOCKET_ERROR。
- 注释：进程可以多次调用 WSAStartup 函数，对于该函数的每一次成功调用，都需要有一次 WSACleanup 相对应。只有最后的 WSACleanup 调用才会进行真正的结束处理工作，其他都只是将 Ws2_32.dll 的内部计数器减一。

17.7　简单的客户端程序

在本章开头，我们说过客户端编程与服务器编程相比较为简单，因此本书侧重于讨论服务器的设计和实现。但是，这并不是说客户端不重要，为了让读者对客户端的工作机制有一定的了解，并且能熟练掌握上文中给出的 Winsock API，本节专门对客户端编程进行讨论。

对于客户端和服务器的区别将在第 18 章作详细分析，目前读者只需要知道：发起通信连接的称为客户端，而等待呼叫请求的称为服务器，并且对于 17.5.1 节讨论的 3 种套接口来说，服务器/客户端的概念通常都是针对 UDP 或者 TCP 套接口而言的。

17.7.1　UDP 客户端

事实上，在 17.6.8 节讨论 recvfrom 和 sendto 函数时给出的例子就是 UDP 客户端。一般来说，UDP 服务器必须绑定本地端口，然后接收外来的 UDP 报文并向客户端反馈数据；而客户端只需要创建一个 UDP 套接口，向服务器的知名端口发送 UDP 报文，然后调用 recvfrom 函数接收反馈即可。读者可以很方便地将程序 17.3 改写为 UDP 客户端程序，这里就不再给出示例。

17.7.2　TCP 客户端

通常，TCP 服务器的设计都非常复杂，必须绑定本地的特定端口，监听并接受外来的连接请求，然后再进行数据交互，并且 TCP 服务器通常都需要能同时处理多个并发的客户端连接。相比较，TCP 客户端就显得简单得多，首先创建一个 TCP 套接口，然后调用 connect 函数主动连接服务器，这时就可以使用 send/recv 函数进行数据交互了。

下面给出一个简单的 TCP 客户端程序 MyTelnet，读者可以使用该程序连接任意 TCP 服务器的任意端口，在命令行方式下输入字符串并回车后，所输入的数据将被发送至服务器，

然后客户端会读取服务器的反馈数据并将其显示。

```
*************************** 程序 17.4 MyTelnet ****************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>

SOCKET g_sockClient = INVALID_SOCKET;

void usage();
BOOL WINAPI CtrlHandler(DWORD dwEvent);

int main(int argc, char* argv[])
{
     unsigned long destAddr;
     int nPort;
     if(argc == 2){
          destAddr = inet_addr(argv[1]);
          if(destAddr == INADDR_NONE){
               usage();
               return -1;
          }
          nPort = 23;
     }
     else
          if(argc == 3){
               destAddr = inet_addr(argv[1]);
               if(destAddr == INADDR_NONE){
                    usage();
                    return -1;
               }
               nPort = atoi(argv[2]);
               if(nPort <= 0 || nPort > 65535){
                    usage();
                    return -1;
               }
          }
          else{
```

```
            usage();
            return -1;
        }

    if(!SetConsoleCtrlHandler(CtrlHandler, TRUE)){
        printf("SetConsoleCtrlHandler: %d\n", GetLastError());
        return -1;
    }

    WSADATA wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);

    g_sockClient = socket(AF_INET, SOCK_STREAM, 0);
    if(g_sockClient == INVALID_SOCKET){
        WSACleanup();
        return -1;
    }

    struct sockaddr_in to;
    memset(&to, 0, sizeof(to));
    to.sin_addr.s_addr = destAddr;
    to.sin_family = AF_INET;
    to.sin_port = htons(nPort);
    printf("connecting %s:%d......", inet_ntoa(to.sin_addr), nPort);
    if(connect(g_sockClient, (struct sockaddr *) &to, sizeof(to)) ==
       SOCKET_ERROR){
        if(g_sockClient != INVALID_SOCKET)
            closesocket(g_sockClient);
        printf("Failed.(connect %d)\n", WSAGetLastError());
        WSACleanup();
        return -1;
    }
    else
        printf("Successfully.\nINPUT:\n");

    char bufa[83];
    char bufb[1000];
    fd_set readSet;
    struct timeval tv;
```

```
    int ret, len;

    while(1){
        memset(bufa, 0, 83);
        gets(bufa);
        len = strlen(bufa);
        if(len > 80) len = 80;
        bufa[len] = '\r';
        bufa[len + 1] = '\n';
        bufa[len + 2] = 0;
        ret = send(g_sockClient, bufa, strlen(bufa), 0);
        if(ret == SOCKET_ERROR){
            printf("send: %d\n", WSAGetLastError());
            break;
        }

        FD_ZERO(&readSet);
        FD_SET(g_sockClient, &readSet);
        tv.tv_sec = 3;
        tv.tv_usec = 0;
        ret = select(0, &readSet, NULL, NULL, &tv);

        if(ret == SOCKET_ERROR){// CASE 1: select Error
            printf("select: %d\n", WSAGetLastError());
            break;
        }
        if(ret == 0){// CASE 2: select Timeout
            printf("Timeout, No Response From Server.\n");
            break;
        }
        if(FD_ISSET(g_sockClient, &readSet)){// CASE 3: select OK
            memset(bufb, 0, 1000);
            ret = recv(g_sockClient, bufb, 1000, 0);
            if(ret == SOCKET_ERROR){
                printf("recv: %d\n", WSAGetLastError());
                break;
            }
            else
                printf("%s\n", bufb);
```

```
            }

        }
        if(g_sockClient != INVALID_SOCKET)
            closesocket(g_sockClient);
        WSACleanup();
        printf("Stopped.\n");
        return 0;
}

void usage()
{
        printf("usage:\tmytelnet x.x.x.x port\t\t\t(0 < port < 65535)\n");
}

BOOL WINAPI CtrlHandler(DWORD dwEvent)
{
        switch(dwEvent){
        case CTRL_C_EVENT:
        case CTRL_LOGOFF_EVENT:
        case CTRL_SHUTDOWN_EVENT:
        case CTRL_CLOSE_EVENT:
            printf("Stopping......\n");
            closesocket(g_sockClient);
            g_sockClient = INVALID_SOCKET;
            break;
        default:
            return FALSE;
        }

        return TRUE;
}
***************************************************************************
```

在命令行方式下输入命令“MyTelnet 202.119.9.99 7”，即连接 202.119.9.99 的 TCP-ECHO 服务端口。程序输出如图 17.7 所示。

该程序向读者演示如何编写一个简单的 TCP 客户端程序，以及 TCP 客户端的基本工作流程。大部分函数调用在前面都已经学过，下面对程序源码进行简单的介绍：



图 17.7　MyTelnet 程序输出

第 1 行 pragma 注释，指示连接器（linker）查找并使用 ws2_32.lib 静态库。如果不使用这种编码方式，那么也可以选择在项目设置的 Link 选项中加入 ws2_32.lib，如图 17.8 所示。

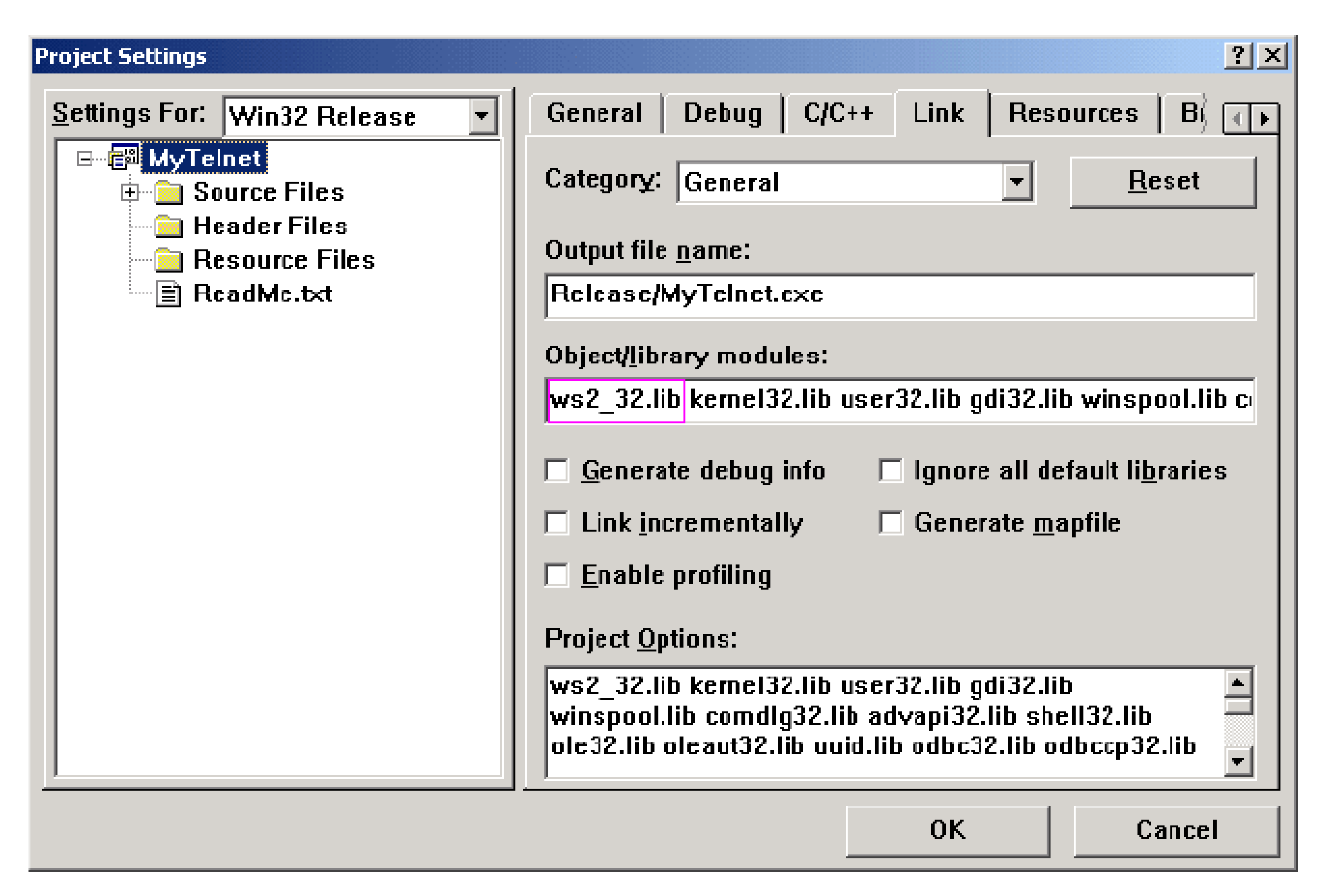


图 17.8　ws2_32.lib 的设置

第 4 行声明了套接口类型的全局变量 g_sockClient。

第 9～35 行读取输入参数目标 IP 和端口，分别保存在 destAddr 和 nPort 变量中。

第 36～39 行调用 SetConsoleCtrlHandler 函数为 MyTelnet 设置某些特殊事件的响应函数，这些特殊事件有 CTRL_C_EVENT、CTRL_CLOSE_EVENT 等。

第 40～41 行调用 WSAStartup 函数，初始化 winsock。

第 42～46 行创建客户端的 TCP 套接口 g_sockClient。

第 47～51 行填充 INET 地址结构 to，即设定所需连接的 TCP 服务器的 IP 地址和端口。

第 53～61 行根据地址 to，连接 TCP 服务器。

第 68～105 行循环体。从控制台读取用户输入数据，发送给服务器，然后接收服务器的反馈并打印。

- 69～75 行从控制台（console）输入读取一行数据，加以处理后保存在 bufa 字符数组中。
- 76～80 行调用 send 函数，将用户输入数据 bufa 发送给服务器。
- 81～104 行读取服务器的反馈数据并将其在控制台窗口输出。该段代码使用了 select

函数进行超时控制，关于 select 的分析与使用参见 18.2.1 节的程序 18.1。

第 106～107 行调用 closesocket 函数关闭客户端套接口 g_sockClient。

第 108 行调用 WSACleanup 函数，结束 winsock 的使用。

第 116～129 行控制台事件的响应函数。如果 MyTelent 接收到 CTRL_C_EVENT、CTRL_LOGOFF_EVENT、CTRL_SHUTDOWN_EVENT 或者 CTRL_CLOSE_EVENT 事件，那么关闭客户端套接口 g_sockClient（将导致主程序中关于 g_sockClient 的 winsock 函数调用出错，从而退出循环），并将其赋值为 INVALID_SOCKET。

在本节中，我们借 MyTelnet 程序复习了常用的 winsock API 函数的使用，并且熟悉了 TCP 客户端的工作流程和机制，此外该程序也为从第 18 章起介绍的服务器编程提供了一个简单的调试工具。总的来说，客户端相对于服务器来说还是非常简单的，读者将会很快发现这一点。

第 18 章　客户－服务器模型

在第 17 章中我们介绍过，很多网络应用系统采用的都是客户－服务器模型，简写为 C/S 模型。事实上，这种应用架构读者应该非常熟悉，例如 IE 浏览器和 Web 服务器之间、Foxmail 和邮件服务器之间等。知名服务器与客户端之间的数据交互按照一定的公开标准进行，从 TCP/IP 体系结构来说属于应用层协议。因此，一方面，我们用同一个浏览器能看到所有 Web 服务器上的页面；另一方面，不管是使用 IE，还是 Netscape 浏览器，都能打开同一个 Web 服务器上的网页。与之相比较，我们自己开发的客户/服务器的差别仅仅在于采用的是私有的应用协议而不是公开标准，在结构上两者完全相同。

18.1　基 本 模 型

图 18.1 展示了最简单的客户－服务器模型，服务器从客户端接收请求，经过请求处理后向客户端返回应答。如何区分客户和服务器，首先要明确的是客户－服务器都是软件，是运行在各种系统下的程序，而不是硬件；两者区分的主要依据是谁发起了通信连接请求，一般来说，发起通信连接的称为客户端，而等待呼叫请求的称为服务器。

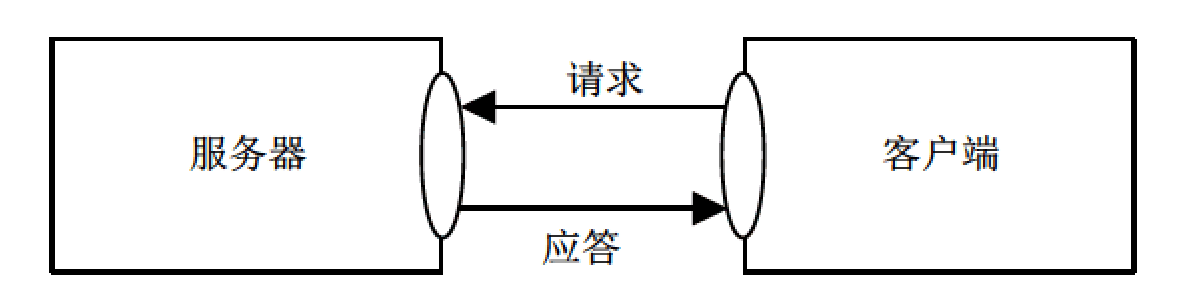


图 18.1　客户－服务器模型

18.1.1　面向连接与无连接

服务器与客户端之间的数据通信是由底层的计算机网络通信协议 TCP/IP 来提供的，TCP/IP 协议族为应用层服务提供了两种传输层服务：面向连接的 TCP 协议和无连接的 UDP 协议。因此，在开发自己的 C/S 系统时，首先要决定的就是选择面向连接的传输协议，还是无连接的传输协议。

TCP 与 UDP 之间的差别是很明显的：UDP 是一种非常简单的传输层协议，它并不能保证数据最终到达目的地；与此相比，TCP 提供可靠的传输服务，确认、超时和重传机制确保了应用层数据会被可靠地传送到目的地，同时 TCP 还提供流量控制。

采用 UDP 还是采用 TCP 并没有一定的标准，通常只有在下面几种情况下采用 UDP 协议：

（1）C/S 之间的数据交互非常简单，如标准服务 daytime 和 echo。

（2）与数据丢失相比，系统更不能容忍超时重传所带来的时间耗费，典型的应用是音视频系统。

（3）系统架构要求采用组播或者广播报文的方式。

除此之外，如果要在系统中采用 UDP 协议，那么就必须设计良好的能保证数据可靠传输的应用层协议；要达到这个目标，确认、超时重传和流量控制或许都是必需的，而这正是 TCP 协议所提供的。

18.1.2　并发和迭代

通常情况下，服务器需要同时向多个客户提供服务，而客户端通常只与一个服务器联系（一个常见的例外是 WEB 浏览器）。我们把能同时接受多个客户连接的服务器称为并发服务器（Concurrent Server），而把一次只能服务一个客户的称为迭代服务器（Iterative Server）。

迭代服务器通常用于提供一些简单的服务，如 daytime、echo 等。对于复杂服务的提供来说，长时间地停顿在为一个客户的服务上而拒绝其他客户是不可取的。

通常，迭代服务器都采用面向无连接的 UDP 协议，而并发服务器采用 TCP 协议，当然这也不是绝对的，在第 20、21 章将加以深入的讨论。

18.2　Winsock I/O 模型

为了开发自己的客户－服务器系统，必须先了解 Winsock 的 I/O 模型。与 I/O 模式概念不同，I/O 模型讨论的是在软件系统层面上对套接口上的 I/O 进行管理及处理的方式。常用的 Winsock I/O 模型有 5 种：select、WSAAsyncSelect、WSAEventSelect、重叠 I/O 以及 I/O 完成端口。

18.2.1　I/O 复用—select

Windows 的 select 函数由 Berkeley Sockets 继承而来，当调用 select 函数时，系统会阻塞在该函数上直到超时或者预设定的某个 I/O 条件（如套接口上有数据可读）得到满足，此时可以进行相应的 I/O 操作（如读数据）并能立即得到结果。

图 18.2 演示了在一个流套接口上直接进行阻塞 recv 操作（左）和先 select 再 recv（右）的流程。两者相比较，采用 select 事实上还多进行了一次函数操作，需要额外的开销。那么为什么要采用 select 模型呢？与普通的阻塞模式相比，该模型的优势主要体现在它能同时判断多个套接口（套接口集合，fd_set）的多种 I/O 状态（read、write、exception）。一次成功的 select 返回，表明至少有一个套接口、满足了一个 I/O 状态要求。具体是哪一些套接口，哪一些 I/O 条件得到了满足需要进行检测，这也是称 select 为 I/O 复用的原因。

select 函数定义如下：

```
int select(int nfds,
  fd_set FAR *readfds, fd_set FAR *writefds, fd_set FAR *exceptfds,
```

```
    const struct timeval FAR *timeout
);
```

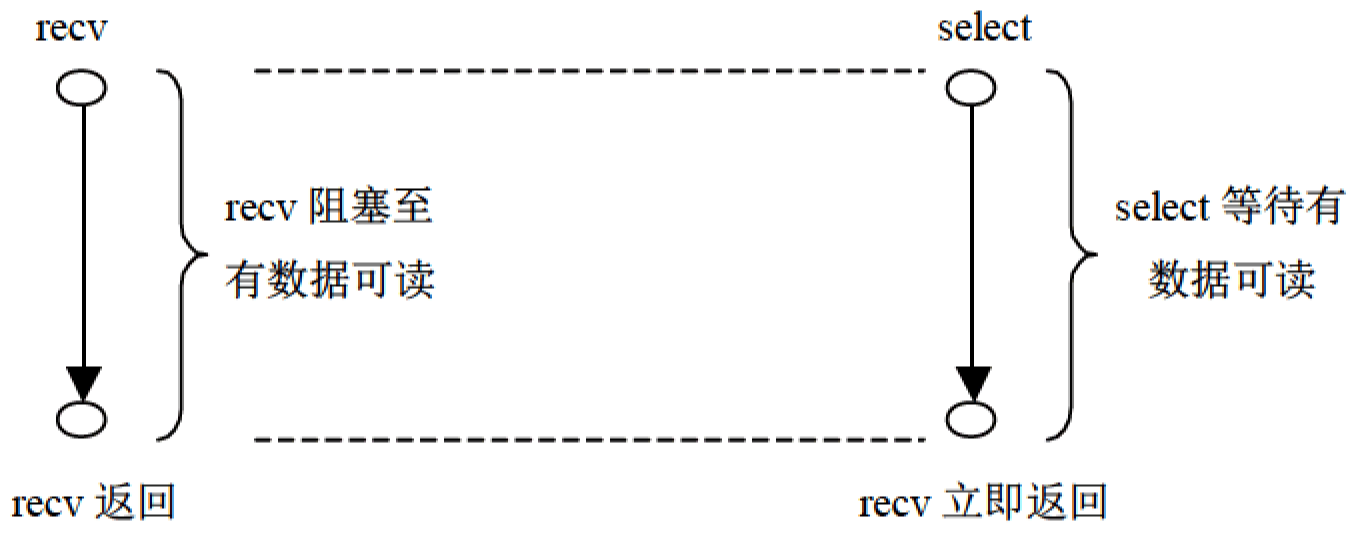


图 18.2　select 的使用

其中第一个参数 nfds 属于输入参数[IN]，在伯克利套接口 API 中表示被测试的套接口描述字的个数，它的值应该被设置为需要测试的最大描述字加一。该参数在 Winsock 中已经被忽略，它的存在仅仅是为了保持兼容性。

第 2～4 个参数 readfds、writefds 和 exceptfds 均是描述字集合（fd_set）结构指针，属于值——结果（输入输出，[INOUT]）类型参数。在伯克利套接口 API 中可将 select 作计时器使用，在 Winsock 中这三个参数不能同时为空指针，否则 select 将返回 10022 错误码（WSAEINVAL - Invalid argument）。

fd_set 结构在很多 Winsock 函数中都会用到，该结构体定义如下：

```
typedef struct fd_set {
  u_int    fd_count;                  // how many are SET?
  SOCKET   fd_array[FD_SETSIZE];      // an array of SOCKETs
} fd_set;
```

其中 fd_count 表示集合中套接口描述字的个数，fd_array 是数组形式的描述字集合。我们一般不会直接对 fd_set 进行操作，套接口 API 中提供了一整套的 FD_XXX 宏定义来完成各种操作任务，这些宏定义如下：

❒ FD_ZERO(*set)

该宏通过将 set 的 fd_count 域置为 0 来完成对该 fd_set 的初始化。

❒ FD_SET(s, *set)

将套接口描述字 s 加入集合 set。

❒ FD_ISSET(s, *set)

检查套接字 s 是否在 set 中，如果在返回非 0 值，否则返回 0。

❒ FD_CLR(s, *set)

从集合 set 中去掉套接字 s。

第 5 个参数 timeout 为输入参数[IN]，该参数以 TIMEVAL 结构的形式告诉 select 函数需要等待的时间。其中，tv_sec 字段是以秒为单位的时间值；tv_usec 字段是以毫秒为单位的时间值。如果该参数值设为 NULL，那么 select 将以阻塞模式工作；如果设置为（0，0），那

么 select 会立即返回，由此可以对套接口状态进行“轮询”，但出于对性能方面的考虑，一般不会这么做。TIMEVAL 结构定义如下：

```
struct timeval {
    long    tv_sec;        // seconds
    long    tv_usec;       // and microseconds
};
```

select 函数返回值有 3 种情况。

- SOCKET_ERROR：表明 select 函数调用出错，这时可以使用 WSAGetLastError 函数来获得错误码。
- 0：表明 select 等待超时，如果第 5 个参数为 NULL，那么不会出现该返回值的情况。
- 其余均为正常返回，说明下列情况中至少有一种发生了，并且返回值表示了满足 I/O 操作的套接口数。

（1）readfds

- 调用了 listen 函数，并且一个 TCP 连接正等待接受（三步握手已完成），这时调用 accept 将成功。
- 有数据可读（包括带外数据，如果开启了 SO_OOBINLINE 选项）。
- TCP 连接被关闭、复位或者中止。

（2）writefds

- 数据可以被发出。
- 内核正处理一个非阻塞的 connect，并且连接成功。

（3）exceptfds

- 内核正处理一个非阻塞的 connect，但是连接失败。
- 有带外数据可读，但是未开启 SO_OOBINLINE 选项。

下面给出一个简单的例子来说明 select 的使用：

```
********************************* 程序 18.1 SelectModel ********************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>

int main(int argc, char* argv[])
{
WSAData wsaData;
WSAStartup(WINSOCK_VERSION, &wsaData);
// 创建流套接口 sockListen
SOCKET sockListen = socket(AF_INET, SOCK_STREAM, 0);
// 将 sockListen 绑定到本地 IP 的 9999 端口
```

```
struct sockaddr_in addr;
int len = sizeof(addr);
memset(&addr, 0, sizeof(addr));
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_family = AF_INET;
addr.sin_port = htons(9999);
if(bind(sockListen, (struct sockaddr *) &addr, len) == SOCKET_ERROR){
    printf("bind: %d\n", WSAGetLastError());
    goto FINISH;
}
// sockListen 进入监听状态
if(listen(sockListen, 1) == SOCKET_ERROR){
    printf("listen: %d\n", WSAGetLastError());
    goto FINISH;
}

SOCKET sockSvr;
int ret;
char buf[1200];
fd_set readSet;
timeval tv;
memset(&tv, 0, sizeof(tv));
tv.tv_sec = 8;
while(1){
    // 接受连接，返回 sockSvr
    sockSvr = accept(sockListen, NULL, NULL);
    if(sockSvr == INVALID_SOCKET)
        break;
    // select 操作
    FD_ZERO(&readSet);
    FD_SET(sockSvr, &readSet);
    ret = select(0, &readSet, NULL, NULL, &tv);
    // CASE 1: select Error
    if(ret == SOCKET_ERROR){
        printf("select: %d\n", WSAGetLastError());
        closesocket(sockSvr);
        break;
    }
    // CASE 2: select Timeout
    if(ret == 0){
```

```
            closesocket(sockSvr);
            continue;
        }
        // CASE 3: select OK
        if(FD_ISSET(sockSvr, &readSet)){
            memset(buf, 0, 1200);
            ret = recv(sockSvr, buf, 1200, 0);
            if(ret == SOCKET_ERROR){
                printf("recv: %d\n", WSAGetLastError());
                closesocket(sockSvr);
                continue;
            }
            send(sockSvr, buf, ret, 0);
            closesocket(sockSvr);
        }
    }

    FINISH:
    closesocket(sockListen);
    WSACleanup();
    return 0;
    }
    ****************************************************************************
****
```

在上面的代码中，首先创建了一个监听本地 9999 端口的 TCP 套接口 sockListen；然后系统阻塞在 accept 函数调用上直到连接到来并返回 sockSvr；sockSvr 等待对方发送数据，并将接收到的数据反射回对方套接口，如果在 8 秒钟内无数据可读那么 select 超时返回；最后关闭 sockSvr，程序循环回至 accept 阻塞调用。

图 18.3 显示了采用 select 模型的基本步骤，总结如下：

（1）使用 FD_ZERO 宏，将感兴趣的套接字集合 fd_set 初始化。

（2）使用 FD_SET 宏，将感兴趣的套接口描述字分配给相应的 fd_set。

（3）调用 select 函数（一般设置超时时间），如果：

- 返回 SOKCET_ERROR，进行相应的错误处理。
- 返回 0，表示 select 超时，根据系统设计要求，可关闭套接口、切断连接或者返回步骤（1），重新进行 select 的相关调用。
- 正常返回 N，说明有 N 个待处理套接口描述字存在，进入步骤（4）。

（4）使用 FD_ISSET，判断哪些套接口有什么样的待处理 I/O 操作。对 FD_ISSET 宏操作返回非 0 值的套接口进行相应的 I/O 操作。全部处理完毕后，根据需要，返回步骤（1）重新进行 select 的相关调用或者退出。

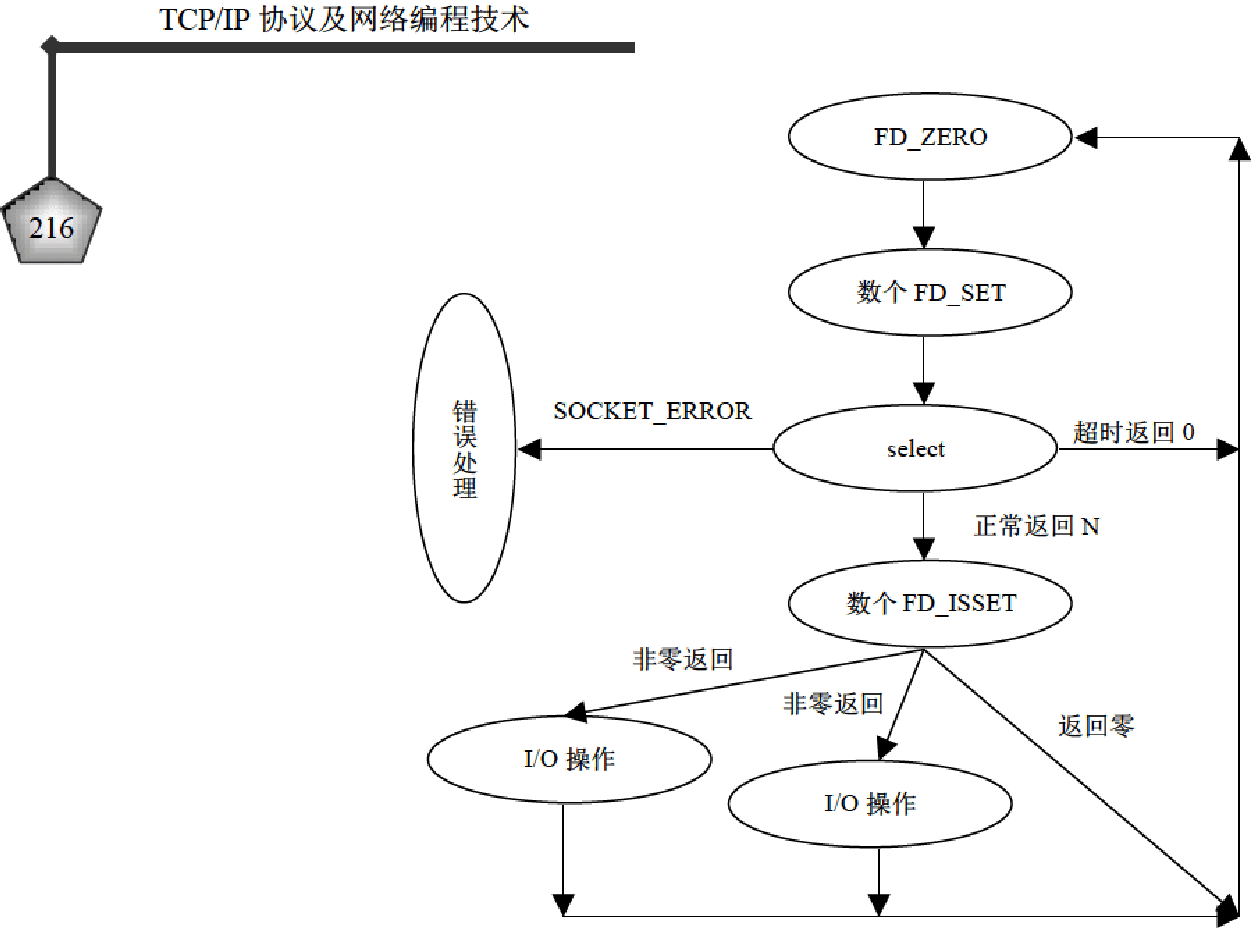


图 18.3　select 模型的应用过程

18.2.2　消息机制——WSAAsyncSelect

WSAAsyncSelect 提供了一种基于 Windows 消息机制的异步 I/O 模型，使用该函数可以为特定套接口上特定网络事件的发生指定系统通知消息。函数定义如下：

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, unsigned int wMsg, long lEvent);
```

全部参数均为输入参数。其中第一个参数 s 是 WSAAsyncSelect 函数操作的套接口描述字，成功的调用 WSAAsyncSelect 会将 s 自动设置为非阻塞模式。

第四个参数 lEvent 用于设定用户所关心的套接口 s 上的网络事件。WSAAsyncSelect 支持的网络事件及其触发条件如表 18.1 所示。lEvent 可由下列事件值通过位或（|）而来，举一个简单的例子，如果关心套接口上的数据可读和关闭事件，那么就有：lEvent = FD_READ | FD_CLOSE。

表18.1　网络I/O事件

事　件　值	含　　义	触发条件
FD_READ	希望得到套接口有数据可读的消息通知	• 调用 WSAAsyncSelect 时已有数据可读 • 有数据到来，并且 FD_READ 消息未发送* • recv 或者 recvfrom 后仍有数据可读

* 如果发送过 FD_READ 消息，那么只有调用下文中将要提到的重新激活函数后才能再次触发 FD_READ 消息的发送，这属于 FD_READ 的第三种触发条件。

续表

事　件　值	含　　义	触发条件
FD_WRITE	希望得到套接口可发送数据的消息通知	• 调用 WSAAsyncSelect 时，可进行 send 或 sendto 操作 • 在 connect 或者 accept 调用后，建立了连接 • 上次的 send 或者 sendto 操作因 WSAEWOULDBLOCK 失败，目前可能会成功 • 在非连接套接口上调用了 bind
FD_OOB	希望得到套接口有带外数据到来的消息通知	• 当调用 WSAAsyncSelect 时已有带外数据可读 • 带外数据到来，并且 FD_OOB 消息未发送 • recv 或 recvfrom 后，仍然有带外数据可读
FD_ACCEPT	希望得到有连接请求待接受的消息通知	• 调用 WSAAsyncSelect 时已有外来连接等待接受 • 有连接请求到来，并且未发送 FD_ACCEPT 消息 • 调用了 accept 函数后，又有连接请求到来
FD_CONNECT	希望得到 connect 或者多点 join 操作完成的消息通知	• 调用 WSAAsyncSelect 时正好有连接完成 • 调用 connect 后，当连接建立时 • 调用 WSAJoinLeaf 后，当 join 操作成功时 • 在非阻塞、面向连接的套接口调用 WSAConnect 或者 WSAJoinLeaf 时返回 WSAEWOULDBLOCK 错误，但是事实上此时网络操作依然在进行着。无论操作最终是失败还是成功，只要结果能确定 FD_CONNECT 消息均会被触发
FD_CLOSE（仅适用面向连接套接口）	希望得到套接口关闭的消息通知	• 调用 WSAAsyncSelect 时，套接口连接被关闭 • 对方套接口正常关闭，并且没有数据待接收（否则会等至所有数据处理完毕） • 本地套接口调用 shutdown（不是 closesocket）主动关闭，对方通知数据发送完毕（如 TCP-FIN），并且没有数据待接收 • 对方中断连接（如发送了 TCP-RST），并且 lParam 包含 WSAECONNRESET 错误值
FD_QOS	希望得到套接口 QOS 状态发生变化的消息通知	• 当 WSAAsyncSelect 调用时，套接口 QOS 发生了变化 • 调用 WSAIoctl（SIO_GET_QOS）改变了套接口 QOS
FD_GROUP_QOS	保留	保留
FD_ROUTING_INTERFACE_CHANGE	希望得到特定方向的路由接口发生改变的消息通知	调用 WSAIoctl（SIO_ROUTING_INTERFACE_CHANGE），本地到达指定目标的网络接口发生了变化
FD_ADDRESS_LIST_CHANGE	希望得到本地地址列表上的套接口协议族发生改变的通知	调用 WSAIoctl（SIO_ADDRESS_LIST_CHANGE），用户进程可绑定的本地地址列表发生了变化

第三个参数 wMsg 是用户为套接口网络事件 lEvent 设定的通知消息，通常 wMsg 指定一个用户定义的消息（WM_USER＋n），例如：

```
#define WM_USER_SERVER      WM_USER+1
```

第二个参数 hWnd 是用户指定的接收系统通知消息 wMsg 的窗体句柄。

如果函数调用成功，WSAAsyncSelect 返回 0；否则返回 SOCKET_ERROR，这时可以调用 WSAGetLastError 来获得具体的错误码。在成功调用 WSAAsyncSelect 后，当有特定网络事件发生时，我们指定的窗体就会收到指定的用户消息。当然，系统并不会为该网络事件连续不断地向用户进程发送通知。事实上，在成功地发送了一次消息后，消息通知机制会暂停工作直到用户进程调用了特定的 Winsock 的 I/O 处理函数，这些函数一方面对网络事件进行相应的 I/O 处理，另一方面会重新激活（re-enable）消息通知机制。网络事件与其对应的重新激活函数如表 18.2 所示。

表18.2　网络事件与重新激活函数

网络事件	Re-enabling 函数
FD_READ	recv, recvfrom, WSARecv 或 WSARecvFrom
FD_WRITE	send, sendto, WSASend 或 WSASendTo
FD_OOB	recv, recvfrom, WSARecv 或 WSARecvFrom
FD_ACCEPT	accept 或 WSAAccept（错误码不能为 WSATRY_AGAIN）
FD_CONNECT	无
FD_CLOSE	无
FD_QOS	WSAIoctl（SIO_GET_QOS）
FD_GROUP_QOS	Reserved
FD_ROUTING_INTERFACE_CHANGE	WSAIoctl（SIO_ROUTING_INTERFACE_CHANGE）
FD_ADDRESS_LIST_CHANGE	WSAIoctl（SIO_ADDRESS_LIST_CHANGE）

举一个简单的例子来说明这种重新激活的概念，假设协议缓冲区中有 1000 个字节的数据，那么系统会发送 FD_READ 事件相对应的某个用户消息（假设为 WM_READ）到指定的窗体，在“发送了 WM_READ 消息”至“用户进程调用 recv 函数读取数据”的这段时间内，系统不会再次为该套接口上的该网络事件发送消息。只有当调用了 recv 函数后，该网络事件的消息通知机制才会被再次激活。

同样以上面的情况为例，假如调用 recv 函数只读取了 500 个字节的数据，这时缓冲区里还有 500 字节待读，那么系统会怎么样，用户进程又该怎么处理呢？这就涉及到事件触发的两种机制：水平触发（Level-Triggered）和边缘触发（Edge-Triggered）。FD_READ、FD_OOB 和 FD_ACCEPT 事件的消息发送均是水平触发的，这就意味着如果重新激活函数（如 recv）被调用后引起消息发送的条件仍然满足（缓冲区里还有 500 字节数据），那么系统会再次发送通知消息。因此我们的程序可以是事件驱动的，而不用管操作时到底该读多少数据（不必一次读完，代码例子可见下一节）。边缘触发的有 FD_QOS、FD_GROUP_QOS 等，由于本书不涉及，这里就不再进行讨论。

对同一套接口多次调用 WSAAsyncSelect 函数，那么只有最后一次调用的设置会生效（在此之前的所有 WSAAsyncSelect 和 WSAEventSelect 的设置均失效，WSAEventSelect 函数将在下一节介绍），因此：

```
WSAAsyncSelect(s, m_hWnd, WM_USER_SERVER, FD_READ), 加上
WSAAsyncSelect(s, m_hWnd, WM_USER_ SERVER, FD_CLOSE) 并不等于
```

```
WSAAsyncSelect(s, m_hWnd, WM_USER_ SERVER, FD_READ | FD_CLOSE)
```

如果要取消套接口 s 上 I/O 事件的消息通知，可以进行如下的函数调用：

```
WSAAsyncSelect(s, hWnd, 0, 0)
```

在使用 WSAAsyncSelect 函数设定了套接口 s 上的网络事件（如：RD_READ | FD_CLOSE）及其相应的 Windows 消息（如：WM_USER_SERVER）之后，我们就需要进行消息处理了。以 MFC 编程环境为例，首先设定该消息的处理函数，见斜体部分：

```
BEGIN_MESSAGE_MAP(CXXXDlg, CDialog)
    //{{AFX_MSG_MAP(CXXXDlg)
    ……
    //}}AFX_MSG_MAP
    ON_MESSAGE(WM_USER_SERVER, OnServerMsg)
……
END_MESSAGE_MAP()
```

消息处理函数声明为：

```
afx_msg void OnServerMsg(WPARAM wParam, LPARAM lParam);
```

它的定义为：

```
void CXXXDlg::OnServerMsg(WPARAM wParam, LPARAM lParam)
{
    SOKCET sock = (SOCKET) wParam;
    if(WSAGETSELECTERROR(lParam)){
        ErrorProcess();
        return;
    }

    switch(WSAGETSELECTEVENT(lParam)){
    case FD_READ:
        ReadData(sock); //读数据并进行相应处理
        break;
    case FD_CLOSE:
        Finsish(sock); //套接口关闭工作
        break;
    default:
        break;
    }
}
```

消息处理函数会接收到系统传来的两个输入参数 wParam 和 lParam。其中 wParam 参数指明了发生网络事件的套接口，如果为多个套接口（通常是服务器端由多次 accept 返回的多个已连接套接口）指定了同一个用户消息，那么就需要根据 wParam 来判断到底是哪一个套接口待处理。在 lParam 参数中包含了两方面的信息：lParam 的低字指定了发生的网络事件；lParam 的高字包含了可能出现的错误代码。在上面的 OnServerMsg 函数里用到了两个新的宏



定义：WSAGETSELECTERROR 和 WSAGETSELECTEVENT。事实上，它们的作用就是取高/低字，在 Winsock2.h 中的定义如下。考虑到今后可能的变化和程序的兼容性，我们建议采用宏操作。

```
#define WSAGETSELECTERROR(lParam)       HIWORD(lParam)
#define WSAGETSELECTEVENT(lParam)       LOWORD(lParam)
```

对于 FD_WRITE 事件的处理，需要强调的是当用户进程收到第一个 FD_WRITE 用户消息后就应该认为自己能向套接口写数据，直到发送操作碰到 WSAEWOULDBLOCK 错误时才需要等待下一次 FD_WRITE 通知消息。一般情况下，不需要考虑套接口是否可写的问题，也就不用关心 FD_WRITE 事件。

最后，我们总结一下 WSAAsyncSelect 模型在 MFC 环境下的使用流程：

（1）使用#define 语句定义为套接口网络事件设置的用户消息值，一般为 WM_USER+N 形式。

（2）调用 WSAAsynsSelect 函数，为套接口设定“网络事件－用户消息－消息接收窗体”的对应关系。

（3）在消息接收窗体的代码的消息映射模块中，加入 ON_MESSAGE 宏，设定用户消息的处理函数。

（4）编写用户处理函数，该函数应该首先使用 WSAGETSELECTERROR 宏判断是否有错误发生；然后根据 wParam 值了解是哪一个套接口上发生了网络事件从而引起该用户消息被系统发送；最后使用 WSAGETSELECTEVENT 宏来了解所发生的网络事件，从而进行相应的处理。

18.2.3　事件机制—WSAEventSelect

WSAEventSelect 与 WSAAsyncSelect 模型类似，同样是系统为套接口上的网络事件 FD_XXX 向用户进程提供通知服务。当收到通知时，进程就可以进行相应的 I/O 操作并立即返回得到结果。两者的差别是：对于前者，系统发送事件对象通知（Event-Object），并且在内部的网络事件记录中加以记录；对于后者，系统向指定窗体发送用户定义的 Windows 消息。其函数定义如下：

```
int WSAEventSelect(SOCKET s, WSAEVENT hEventObject, long lNetworkEvents);
```

三个参数均为输入参数。其中第一个参数 s 是套接口描述字。第三个参数 lNetworkEvents 是我们感兴趣的套接口网络事件，所有可能的事件值以及值的组合均与 WSAAsyncSelect 函数的 lEvent 参数相同。如果该值设为 0，那么将取消在套接口 s 上的所有网络事件与事件句柄之间的联系设定。此外，无论该值是否为 0，成功的调用 WSAEventSelect 都会将 s 设置为非阻塞模式。

第二个参数 hEventObject 是一个 Winsock 的事件对象。当套接口 s 上发生了 lEvent 中指定的网络事件时，系统会发送 hEventObject 并将其记录在内部事件记录中。在使用 WSAEventSelect 函数之前，首先需要创建 hEventObject。关于 Winsock 事件对象的三个基本操作是 WSACreateEvent、WSAResetEvent 和 WSACloseEvent，它们的作用分别是创建事件

对象、复位事件对象和关闭事件对象。函数定义如下：

- WSAEVENT WSACreateEvent (void)

如果操作成功，函数返回新创建的 WSAEVENT 对象，否则返回 WSA_INVALID_EVENT。

- BOOL WSAResetEvent(WSAEVENT hEvent)

事件对象有两个状态：已触发（signaled）和未触发（nonsignaled）。新创建的事件对象为未触发状态，我们也可以调用 WSAResetEvent 函数来将已触发 hEvent 事件复位为未触发状态，如果操作成功，函数返回 TRUE，否则 FALSE。与此相对应，Winsock2 还提供了 WSASetEvent 函数来将事件对象设置为已触发状态，本书不涉及该函数的使用。在 18.2.4 节介绍重叠操作时，会发现很多 Winsock2 函数操作会自动复位特定的事件对象。

- BOOL WSACloseEvent(WSAEVENT hEvent);

当事件对象使用完毕后，需要调用 WSACloseEvent 将它关闭，函数操作成功则返回 TRUE，否则返回 FALSE。

成功地调用 WSAEventSelect 函数会返回 0，它说明关于套接口 s 上的“网络事件—事件对象”关系已被设定，并且内部网络事件记录也已被清除；否则返回 SOCKET_ERROR。WSAEventSelect 也存在重新激活（re-enable）和水平触发—边缘触发的概念，Winsock2 对它和 WSAAsyncSelect 的处理方式是完全相同的，相关的分析可参照 18.2.2 节。

与 WSAASyncSelect 函数一样，对同一个套接口调用多次 WSAEventSelect 函数，只有最后一次设置会生效，此前无论是 WSAEventSelect 还是 WSAASyncSelect 的设定都会被取消。因此，我们也无法为同一个套接口的不同网络事件设置不同的事件对象。举例来说，下面的代码仅仅起到了为套接口 s 的 FD_WRITE 事件设置 hEventObject2 对象的作用：

```
ret = WSAEventSelect(s, hEventObject1, FD_READ);
ret = WSAEventSelect(s, hEventObject2, FD_WRITE);
```

下面的代码取消了套接口上任意的网络事件与事件对象的联系设定，其中的参数 hEventObject 将被忽略：

```
ret = WSAEventSelect(s, hEventObject, 0);
```

在成功的设定了套接口上的网络事件—事件对象的联系后，可以调用函数 WSAWaitForMultipleEvents 来等待或者轮询事件对象的状态（是否已触发），并且可以使用函数 WSAEnumNetworkEvents 以获得内部网络事件记录内容来判断到底是哪一个网络事件发生了。这两个函数的定义如下：

```
DWORD WSAWaitForMultipleEvents(
    DWORD cEvents, const WSAEVENT FAR *lphEvents,
    BOOL fWaitAll, DWORD dwTimeout, BOOL fAlertable
);
```

所有的参数都是输入参数。其中 lphEvents 是网络事件对象的数组指针。cEvents 是该数组中的事件对象的数目，最大为 WSA_MAXIMUM_WAIT_EVENTS，至少为一。fWaitAll 表示等待的类型，如果为 TRUE，函数将等待 lphEvents 数组中的所有事件对象的状态都变为



已触发，否则等待任意事件对象变为已触发。dwTimeout 表示等待的超时时间（以微秒为单位），如果为 0，函数操作将立即返回；如果为 WSA_INFINITE，函数将阻塞直至事件对象状态满足要求；当函数操作超时时，无论 fWaitAll 参数条件是否已满足，函数均会返回。fAlertable 用于设置当有 I/O 完成例程（completion routine）排队等待执行时，WSAWaitForMultipleEvents 函数是否需要返回，该参数的使用在本书中不涉及，设置为 FALSE。

如果函数发生错误，返回 WSA_WAIT_FAILED，否则返回以下几种结果。

（1）WSA_WAIT_EVENT_0 ~ WSA_WAIT_EVENT_0+（cEvents-1）：如果 fWaitAll 为 TRUE，表示所有事件对象均已触发；否则返回值减去 WSA_WAIT_EVENT_0 即表示事件对象数组中的已被触发的事件对象的数组下标。

（2）WAIT_IO_COMPLETION：一个或多个 I/O 完成例程已排队待执行。

（3）WSA_WAIT_TIMEOUT：函数操作超时，fWaitAll 参数指定的事件触发要求未得到满足。

```
int WSAEnumNetworkEvents(
      SOCKET s,
      WSAEVENT hEventObject,
      LPWSANETWORKEVENTS lpNetworkEvents
);
```

如果将多个网络事件（如 FD_READ | FD_CLOSE）与同一个事件对象联系起来，那么当该事件对象被触发时，就需要使用 WSAEnumNetworkEvents 函数从内部的网络事件记录中找出到底是哪一个网络事件触发了事件对象。这相当于 WSAAsyncSelect 模型中消息处理函数的 WSAGETSELECTEVENT(lParam)的作用。

其中前两个参数为输入参数，分别用于设定感兴趣的套接口和事件对象。第三个参数是输出参数，它是一个指向 WSANETWORKEVENTS 结构体的指针。该结构定义如下：

```
typedef struct _WSANETWORKEVENTS {
  long     lNetworkEvents;
  int      iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, *LPWSANETWORKEVENTS;
```

它包含了两部分的重要信息：lNetworkEvents 表示 FD_XXX 类型的网络事件值；iErrorCode 是可能的错误码的数组。在 Winsock2.h 中定义了很多类似于 FD_XXX_BIT 的常量值（0 ～ FD_MAX_EVENTS-1），例如：#define FD_READ_BIT 0 等，其中常用的有 FD_READ_BIT、FD_WRITE_BIT、FD_ACCEPT_BIT、FD_CONNECT_BIT、FD_CLOSE_BIT 等，可以直接读 iErrorCode[FD_XXX_BIT]来获取可能的套接口输入错误。

WSAEnumNetworkEvents 如果没有错误发生，函数返回 0；否则返回 SOCKET_ERROR。

我们已经介绍完了 WSAEventSelect 模型涉及到的大部分操作，下面提供一个简单的例子来说明它的使用：

```
*********************** 程序18.2 WSAEventSelect_Model *********************
#pragma comment(lib, "ws2_32.lib")
```

```
#include <STDIO.H>
#include <WINSOCK2.H>

int main(int argc, char* argv[])
{
    WSAData wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);

    SOCKET sockListen = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    int len = sizeof(addr);
    memset(&addr, 0, sizeof(addr));
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(9999);
    if(bind(sockListen, (struct sockaddr *) &addr, len) == SOCKET_ERROR){
        printf("bind: %d\n", WSAGetLastError());
        closesocket(sockListen);
        WSACleanup();
    }

    if(listen(sockListen, 1) == SOCKET_ERROR){
        printf("listen: %d\n", WSAGetLastError());
        closesocket(sockListen);
        WSACleanup();
    }

    SOCKET sockSvr;
    int ret;
    char buf[1200];

    WSAEVENT ev = WSACreateEvent();// 创建事件对象
    if(ev == WSA_INVALID_EVENT){
        printf("WSACreateEvent: %d\n", WSAGetLastError());
        closesocket(sockListen);
        WSACleanup();
    }
    WSANETWORKEVENTS evInfo;

    while(1){// 主循环：接受连接—交互—断开连接—接受连接......
        // 接受连接，返回 sockSvr
        sockSvr = accept(sockListen, NULL, NULL);
```

```
        if(sockSvr == INVALID_SOCKET){// 直接退出主循环
            printf("accept: %d\n", WSAGetLastError());
            break;
        }
        // 设定关注的网络事件和事件对象之间的联系
        ret = WSAEventSelect(sockSvr, ev, FD_READ | FD_CLOSE);
        if(ret == SOCKET_ERROR){// 直接退出主循环
            printf("WSAEventSelect: %d\n", WSAGetLastError());
            closesocket(sockSvr);
            break;
        }

        bool bClosed = false;
        while(!bClosed){// 针对每个连接的服务循环
            // 将ev复位为未触发
            WSAResetEvent(ev);

            // 等待ev触发，超时设为8秒
            ret = WSAWaitForMultipleEvents(1, &ev, false, 8000, FALSE);
            if(ret == WSA_WAIT_FAILED){// Error，退出服务循环
                printf("WSAWaitForMultipleEvents: %d\n", WSAGetLastError());
                bClosed = true;
                break;
            }
            if(ret == WSA_WAIT_TIMEOUT){// Timeout，退出服务循环，主动关闭连接
                bClosed = true;
                break;
            }
            if(ret != WSA_WAIT_EVENT_0){// we set only one EvObj, so there must be error
                bClosed = true;
                break;
            }
            // 获取具体的网络事件信息
            memset(&evInfo, 0, sizeof(evInfo));
            ret = WSAEnumNetworkEvents(sockSvr, (&ev)[ret - WSA_WAIT_EVENT_0], &evInfo);
            if(ret == SOCKET_ERROR){
                printf("WSAEnumNetworkEvents: %d\n", WSAGetLastError());
                bClosed = true;
                break;
            }
            // 检查是否有错误发生
```

```
80                  if(evInfo.iErrorCode[FD_READ_BIT]!=0 || evInfo.iErrorCode[FD_
                       CLOSE_BIT]!=0){
81                      printf("ErrorCode: %d", evInfo.iErrorCode[FD_READ_BIT]==0 ?
                        evInfo.iErrorCode[FD_CLOSE_BIT] : evInfo.iErrorCode[FD_READ
                        _BIT]);
82                      bClosed = true;
83                      break;
84                  }
85                  // 如果没有错误发生，检索发生的网络事件
86                  if(evInfo.lNetworkEvents & FD_READ){// recv and echo
87                      memset(buf, 0, 1200);
88                      recv(sockSvr, buf, 1200, 0);
89                      printf("recvd: %s\n", buf);
90                      send(sockSvr, buf, 1200, 0);
91                  }
92                  if((!bClosed) && (evInfo.lNetworkEvents & FD_CLOSE)){// 被动关闭连接
93                      printf("Peer closed!\n");
94                      bClosed = true;
95                  }
96              }// 服务循环结束
97              closesocket(sockSvr);
98          }// 主循环结束
99          WSACloseEvent(ev);
100
101         closesocket(sockListen);
102         WSACleanup();
103         return 0;
104 }
```

****

与 18.1 完全相同，本程序实现了简单的 Echo 服务。结合该程序，我们对 WSAEventSelect 模型应用的主要流程作如下总结。

（1）调用 WSACreateEvent 函数创建一组 Winsock 事件对象（29～34 行）。

（2）调用 WSAEventSelect 函数在感兴趣的套接口（假设为 sockSvr，可能有多个）上建立事件对象与网络事件（假设为 FD_READ | FD_CLOSE）之间的联系（44～50 行）。

（3）调用 WSAWaitForMultipleEvents 等待事件对象的触发，其返回值一般有 3 种情况（56～70 行）。

- WSA_WAIT_FAILED，函数调用出错，进行相应的错误处理。
- WSA_WAIT_TIMEOUT，等待网络事件超时，一般选择断开连接或者重新开始等待，跳至步骤（8）。
- 正常返回值。当调用 WSAWaitForMultipleEvents 时的参数 fWaitAll 为 false，那么该返回值与 WSA_WAIT_EVENT_0 的差值表示被触发的事件对象在数组中的下标；

如果 fWaitAll 为 true，说明数组中的所有事件均已触发；针对每一个被触发事件（假设为 ev），进入步骤（4）。

（4）将 WSANETWORKEVENTS 结构体（假设为 evInfo）清 0，然后以 sockSvr、ev 以及&evInfo 为参数调用函数 WSAEnumNetworkEvents，以获得相应的网络事件信息，其结果保存在 evInfo 中（71～78 行）。

（5）根据 evInfo 的 iErrorCode 域判断是否有特定的错误发生（79～84 行）。

（6）将 evInfo 的 lNetworkEvents 域分别和 FD_READ、FD_CLOSE 相比（&），如果结果不为 0，那么说明该网络事件发生了（85～95 行）。

（7）对发生的网络事件进行相应的 I/O 处理。值得注意的是，根据 18.2.2 节中对于水平触发的分析，我们在获知 FD_READ 事件后只需要简单地调用 recv(sockSvr, buf, 1200, 0)，而不需要考虑究竟有多少数据可读，是否多于 1200 字节。

（8）调用 WSAResetEvent 将 ev 复位为未触发（可省略）；回至步骤（3）。

18.2.4　重叠 I/O 模型

重叠（Overlapped）I/O 是一种比较复杂的 I/O 模型，与前三种模型都围绕某个特定函数相比，重叠 I/O 涉及到很多套接口 I/O 操作函数。因此在介绍它之前，首先对这些函数进行简单的描述。

在第 17 章中介绍了基本的套接口 I/O 操作函数——recv/send 和 recvfrom/sendto，事实上，Winsock2 还提供了一组对应的以 WSA 起头的函数，WSARecv/WSASend 和 WSARecvfrom /WSASendto。除了函数参数的区别之外，它们之间最大的差别就在于后者提供了套接口上的重叠 I/O 操作（还有两点重要的区别是分散/聚合类型的 I/O 操作和 lpFlags 参数改为输入输出类型并结合以 MSG_PARTIAL 标志的使用，这些都不是本书关注的重点）。

首先介绍几个相关的数据结构：

```
（1）typedef struct __WSABUF {
    u_long      len;
    char FAR    *buf;
} WSABUF, FAR * LPWSABUF;
```

与 recv/send 等函数直接操作 char 指针类型的用户缓冲区不同，WSARecv/WSASend 和 WSARecvfrom/WSASendto 等函数操作的对象是 WSABUF——一种新定义的用户缓冲区结构。其中 len 表示用户进程提供的缓冲区的大小，buf 是指向缓冲区的指针。在使用 WSABUF 之前，必须首先为 buf 指针分配空间，然后将该缓冲区的大小赋给 len。同时，在调用这些 I/O 函数时，还允许指定 dwBufferCount 不为 1，即能同时使用多个 WSABUF 用户缓冲区，称为分散/聚合（scatter/gather）I/O 操作。

```
（2）typedef struct _WSAOVERLAPPED {
    DWORD        Internal;
    DWORD        InternalHigh;
    DWORD        Offset;
    DWORD        OffsetHigh;
```

```
    WSAEVENT    hEvent;
} WSAOVERLAPPED, *LPWSAOVERLAPPED;
```

WSAOVERLAPPED 是从 Win32 的 OVERLAPPED 结构移植而来的，与后者完全兼容。它向用户提供了一种在重叠 I/O 操作的初始投递和 I/O 的完成之间进行联系的机制。其中前四个参数均为保留域，由系统内部使用。我们只需要将 hEvent 设置为特定的 WSAEVENT 对象即可。

```
(3) typedef void (CALLBACK * LPWSAOVERLAPPED_COMPLETION_ROUTINE)(
    DWORD dwError,
    DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwFlags
);
```

LPWSAOVERLAPPED_COMPLETION_ROUTINE 是 Winsock2 中重叠 I/O 操作的回调函数指针，当 I/O 操作完成时系统会自动调用由该指针所指向的完成函数。四个参数中 dwError 表示由 lpOverlapped 指定的重叠操作的完成状态；cbTransferred 表示重叠操作过程中传输的字节数；lpOverlapped 参数包含调用 I/O 操作时的 WSAOVERLAPPED 结构信息；dwFlags 参数分为两种情况，对于 WSARecv/WSARecvfrom 函数来说，如果接收操作能立即结束那么 dwFlags 包含了调用接收函数时的 lpFlags 参数的信息，对于 WSASend/WSASendto 函数来说该参数暂未使用，系统会设置为 0。

下面列出了四个 Winsock2 中的重叠 I/O 操作函数的定义：

```
(1) int WSARecv(
        SOCKET s,
        LPWSABUF lpBuffers,
        DWORD dwBufferCount,
        LPDWORD lpNumberOfBytesRecvd,
        LPDWORD lpFlags,
        LPWSAOVERLAPPED lpOverlapped,
        LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
    );
(2) int WSASend(
        SOCKET s,
        LPWSABUF lpBuffers,
        DWORD dwBufferCount,
        LPDWORD lpNumberOfBytesSent,
        DWORD dwFlags,
        LPWSAOVERLAPPED lpOverlapped,
        LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
    );
(3) int WSARecvFrom(
        SOCKET s,
        LPWSABUF lpBuffers,
        DWORD dwBufferCount,
```

```
        LPDWORD lpNumberOfBytesRecvd,
        LPDWORD lpFlags,
        struct sockaddr FAR *lpFrom,
        LPINT lpFromlen,
        LPWSAOVERLAPPED lpOverlapped,
        LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
    );
 (4) int WSASendTo(
        SOCKET s,
        LPWSABUF lpBuffers,
        DWORD dwBufferCount,
        LPDWORD lpNumberOfBytesSent,
        DWORD dwFlags,
        const struct sockaddr FAR *lpTo,
        int iToLen,
        LPWSAOVERLAPPED lpOverlapped,
        LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
    );
```

这组函数的参数、返回值以及系统的处理都基本相同，下面统一对它们进行分析。

要进行重叠 I/O 操作，首先套接口 s 必须为重叠套接口。可以直接调用 socket 函数创建或者以参数 dwFlags 为 WSA_FLAG_OVERLAPPED 参数调用 WSASocket。下面两行代码完成了同样的套接口创建工作。

```
sock = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
sock = socket(AF_INET, SOCK_STREAM, 0);
```

按照 lpOverlapped 和 lpCompletionRoutine 不同的取值情况，这些 I/O 函数有 3 种工作状态。

- 如果 lpOverlapped 和 lpCompletionRoutine 均设为 NULL，那么系统将这些函数作为普通的非重叠 I/O 操作处理，其表现和相应的发送、接收函数（recv/send、recvfrom/sendto）类似。
- 如果仅 lpCompletionRoutine 为 NULL，那么当调用这些 I/O 函数时 lpOverlapped 结构中的 hEvent 事件会被自动复位为未触发，当重叠 I/O 操作完成时再被设置为已触发。用户进程可以调用 WSAWaitForMultipleEvents 或者 WSAGetOverlappedResult 来等待或者轮询该事件对象的状态。其中 WSAWaitForMultipleEvents 函数在 18.2.3 节已经介绍过，WSAGetOverlappedResult 将在下文介绍。
- 如果 lpCompletionRoutine 不为 NULL，那么 hEvent 参数将被忽略，但是仍然可以被用来向完成例程传递信息。此时如果调用 WSAGetOverlappedResult 应注意 fWait 参数不能设置为 TRUE。

在重叠套接口上进行的重叠数据发送/接收操作将工作在非阻塞模式，因此函数调用能立即返回，其返回值可分为 3 种情况。

- 返回 0，说明 I/O 操作已立即完成并且无错误发生，此时 I/O 完成例程已完成调度准备执行。

- ❒ 返回 SOCKET_ERROR，错误码为 997（WSA_IO_PENDING），说明重叠 I/O 操作已成功初始化，在操作完成后应用进程会得到系统的通知。
- ❒ 返回 SOCKET_ERROR，错误码不是 WSA_IO_PENDING，说明重叠 I/O 操作初始化失败。

在完成重叠 I/O 函数调用后，可以调用 WSAGetOverlappedResult 函数来获取指定套接口上的最近一次重叠操作的结果。其函数定义如下：

```
BOOL WSAGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags
);
```

除了 lpdwFlags、lpcbTransfer 是输出参数外，其余均为输入参数。其中 s 是感兴趣的套接口描述字，lpOverlapped 用于返回在该套接口上进行的最近一次重叠操作时的 WSAOverlapped 结构数据，lpcbTransfer 用于输出重叠操作传输的字节数，lpdwFlags 用于输出操作的参数，fWait 参数指示是否要等待事件对象 lpOverlapped->hEvnet，当且仅当选择了基于事件的完成通知（即 lpCompletionRoutine 为 NULL，lpOverlapped 不为 NULL）时，才能将 fWait 设置为 TRUE。如果无错误发生，WSAGetOverlappedResult 返回 TRUE，说明重叠操作已成功完成；否则返回 FALSE，说明或者重叠操作没有完成，或者重叠操作已完成但发生了错误，或者 WSAGetOverlappedResult 调用时有参数设置错误导致重叠操作的状态无法判断。

下面的例子说明了如何应用重叠操作的事件机制来实现一个简单的 Echo 服务器。

```
************************* 程序 18.3 OverLapped1 ***************************
#pragma comment(lib, "ws2_32.lib")
#include <STDIO.H>
#include <WINSOCK2.H>

int main(int argc, char* argv[])
{
     int ret;
     WSAData wsaData;
     WSAStartup(WINSOCK_VERSION, &wsaData);

     // 创建 WSA_FLAG_OVERLAPPED 属性的套接口
     SOCKET sockListen = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
        WSA_FLAG_OVERLAPPED);
     // 也可以简单地调用 socket 函数
     //SOCKET sockListen = socket(AF_INET, SOCK_STREAM, 0);
```

```
    struct sockaddr_in local;
    memset(&local, 0, sizeof(local));
    local.sin_addr.s_addr = INADDR_ANY;
    local.sin_family = AF_INET;
    local.sin_port = htons(9999);
    bind(sockListen, (struct sockaddr *) &local, sizeof(local));
    listen(sockListen, 1);

    SOCKET sockSvr = accept(sockListen, NULL, NULL);
    closesocket(sockListen);

    // 创建 Winsock 事件对象
    WSAEVENT ev = WSACreateEvent();
    WSAOVERLAPPED ol;
    ZeroMemory(&ol, sizeof(ol));
    // 我们仅需关心 hEvent 域
    ol.hEvent = ev;

    char buf[65];
    WSABUF wsaBuf;
    unsigned long nRecved;
    DWORD flags;
    while(1){
        memset(buf, 0, 65);
        wsaBuf.len = 65;
        wsaBuf.buf = buf;
        flags = 0;
        nRecved = 0;
        // 非阻塞调用，将 sockSvr、wsaBuf 和 ol 联系起来，一般返回错误码
           WSA_IO_PENDING
        // 调用时，ol 中事件对象会被自动复位
        // 当后台接收数据完成，wsaBuf 会被自动填充
        if(WSARecv(sockSvr, &wsaBuf, 1, &nRecved, &flags, &ol, NULL)
           == SOCKET_ERROR){
            ret = WSAGetLastError();
            if(ret != WSA_IO_PENDING){
                printf("WSARecv: %d\n", ret);
                break;
            }
        }
        else if(nRecved == 0){// 对方关闭了连接!!!
            printf("对方关闭了连接!!!\n");
            break;
```

```
53              }

54              // 等待 I/O 结束，获取重叠操作操作结果，判断该 I/O 是否成功
55              if(!WSAGetOverlappedResult(sockSvr, &ol, &nRecved, TRUE,
                   &flags)){
56                  ret = WSAGetLastError();
57                  printf("WSAGetOverlappedResult: %d\n", ret);
58                  break;
59              }
60              if(nRecved != 0){
61                  printf("%s\n", wsaBuf.buf);
62                  send(sockSvr, wsaBuf.buf, nRecved, 0);
63              }
64          }// while(1)

65          WSACloseEvent(ev);
66          closesocket(sockSvr);
67          WSACleanup();
68          return 0;
69      }
*************************************************************************
```

第 10～13 行创建 WSA_FLAG_OVERLAPPED 属性的 TCP 套接口 sockListen。

第 14～20 行绑定本地端口 9999，并调用 listen 函数使 sockListen 处于监听状态。

第 21～22 行在 sockListen 上调用 accept 函数，返回连接套接口 sockSvr 后，将 sockListen 关闭。

第 24～29 行调用 WSACreateEvent 函数创建 Winsock 事件对象 ev，并将其赋值给重叠结构 ol 的 hEvent 域。

第 34～64 行 Echo 服务循环，在 sockSvr 上进行数据的收发。

- 35～53 行，投递 WSARecv 重叠 I/O 操作。由于在重叠 I/O 模型中，该操作为非阻塞调用，因此函数一般都会返回错误码 WSA_IO_PENDING。当 I/O 完成时，接收到的数据会被系统自动填充到 WSARecv 调用时指定的 wsaBuf 缓存中。需要注意的是，在上一章讨论 TCP 套接口 recv 操作时提到，如果函数返回 0，说明对方已断开连接；而对于 WSARecv 来说，由于 WSARecv 函数返回 0 表示操作成功，因此必须特别注意该函数的 lpNumberOfBytesRecvd 参数：如果该参数值为 0，那么也说明对方关闭了连接。如果删除了第一个程序代码中对 nRecved== 0 的判断，那么当对方关闭连接时，WSARecv 函数会返回 0、WSAGetOverlappedResult 返回 TRUE，因而会陷入死循环。
- 54～59 行，调用 WSAGetOverlappedResult 函数，等待套接口 I/O 结束，并获取重叠操作操作结果。
- 60～63 行，在本地显示接收到的数据，并出于程序简洁性的考虑直接调用非重叠 I/O

操作的 send 函数将数据反馈 Echo 给客户端。

第 65 行调用 WSACloseEvent 函数关闭事件对象 ev。

第 66 行关闭服务套接口 sockSvr。

第 67～68 行结束 Winsock 的使用后退出程序。

需要指出的是该 Echo 服务器每次运行都只能为一个客户端服务，当该客户端关闭连接服务器即终止运行，因此程序 18.3 仅能作为演示使用并无实际价值。

在下面的例子中，将演示如何采用完成例程机制实现同样的服务器功能。由于我们希望在主线程中循环调用 WSARecv，而在完成例程对接收到的数据进行处理，这就需要一些小的技巧——在完成例程中，程序只能接收到系统传来的 lpOverlapped 参数而不会得到用于接收数据的用户缓冲区结构地址。解决方法是将重叠结构和用户缓冲区结构绑定在一起，在接收到该重叠结构地址后作简单的运算就能获取用户缓冲区地址（当然也可以把用户缓冲区结构体作为全局变量，但这种方法不具有扩展性）。

最简单的绑定方法如下，此时当得到 ol 的地址，也就得到了 OV_Data 结构体的起始地址，因此在完成例程中作强制转化即可得到 OV_Data 地址。

```
    typedef struct _OV_Data{
    WSAOVERLAPPED ol;
    char    buf[65];
}OV_Data;
```

即 OV_Data *povd = (OV_Data *) lpOverlapped。更普遍的转换公式为:

OV_Data *povd = (OV_Data *)(lpOverlapped – ol 在结构中的偏移量*)。

事实上，微软提供的宏定义 CONTAINING_RECORD 直接提供了这种转换的功能，有:

```
OV_Data *povd = CONTAINING_RECORD (lpOverlapped, OV_Data, ol)
```

CONTAINING_RECORD 有三个输入参数(address, type, field)，分别是指结构实例的某个域的地址，结构名和该结构域的域名。宏的输出是指向结构实例的指针。

另外一个问题，前面曾提到当使用完成例程机制时，重叠结构中的事件对象是被忽略的，那么主线程如何知道重叠 I/O 已完成可以再次进行 WSARecv 操作呢？在例子中采用了一种较简单的方法——SleepEx 调用。该函数有两个参数，第一个参数 dwMilliseconds 是以微秒为单位的休眠（等待）时间，第二个参数 bAlertable 表示是否设置为可唤醒的，如果为 TRUE，那么当有 I/O 完成例程发生，SleepEx 将提前结束并返回 WAIT_IO_COMPLETION。

下面是采用重叠 I/O 模型完成例程机制的 Echo 服务器的源程序，该程序与 18.3 差别不大，就不再作详细分析了。

```
************************** 程序18.4 OverLapped2 **************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>
```

---

* 结构 XYZ（{X, Y, Z}）中域 Y 的偏移量的计算公式为&((XYZ *)0)->Y，使用时需注意类型转换。

```
typedef struct _OV_Data{
    SOCKET   sock;
    WSABUF   wsaBuf;
    char data[65];
    WSAOVERLAPPED ol;
}OV_Data;

SOCKET g_sockSvr = INVALID_SOCKET;

void CALLBACK ProcessData(DWORD dwError, DWORD cbTransferred, LPWSAOVERLAPPED
    lpOverlapped, DWORD dwFlags)
{
    OV_Data *lpovd = CONTAINING_RECORD(lpOverlapped, OV_Data, ol);
    //OV_Data *lpovd = (OV_Data *)((PCHAR)lpOverlapped - (UINT_PTR)(&((OV_Data
    *)0)->ol));

    if(dwError!=0 || cbTransferred==0)
        return;

    printf("%s\n", lpovd->wsaBuf.buf);
    send(g_sockSvr, lpovd->wsaBuf.buf, cbTransferred, 0);
}

int main(int argc, char* argv[])
{
    int ret;
    WSAData wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);
    // 创建WSA_FLAG_OVERLAPPED属性的套接口
    SOCKET  sockListen  =  WSASocket(AF_INET,  SOCK_STREAM,  0,  NULL,  0,
    WSA_FLAG_OVERLAPPED);

    struct sockaddr_in local;
    memset(&local, 0, sizeof(local));
    local.sin_addr.s_addr = INADDR_ANY;
    local.sin_family = AF_INET;
    local.sin_port = htons(9999);
    bind(sockListen, (struct sockaddr *) &local, sizeof(local));

    listen(sockListen, 1);
    g_sockSvr = accept(sockListen, NULL, NULL);
    closesocket(sockListen);

    // 创建Winsock事件对象
    WSAEVENT ev = WSACreateEvent();

    OV_Data ovd;
```

```
    unsigned long nRecved;
    DWORD flags;

    while(1){
        memset(&ovd, 0, sizeof(ovd));
        ovd.ol.hEvent = ev;
        ovd.wsaBuf.buf = ovd.data;
        ovd.wsaBuf.len = 65;
        ovd.sock = g_sockSvr;

        nRecved = 0;
        flags = 0;

       if(WSARecv(g_sockSvr, &(ovd.wsaBuf), 1, &nRecved, &flags, &(ovd.ol),
       ProcessData) == SOCKET_ERROR){
            ret = WSAGetLastError();
            if(ret != WSA_IO_PENDING){
                printf("WSARecv: %d\n", ret);
                break;
            }
        }
        else if(nRecved == 0){// 对方关闭了连接!!!
            printf("对方关闭了连接!!!\n");
            break;
        }

        ret = SleepEx(12000, TRUE);
        if(ret == 0){
            printf("Wait time out!\n");
            break;
        }
        else if(ret == WAIT_IO_COMPLETION)
            continue;
        else
            break;
    }

    WSACloseEvent(ev);
    closesocket(g_sockSvr);
    WSACleanup();
    return 0;
}
************************************************************************
```

18.2.5 I/O 完成端口——IOCP

IOCP 从本质上来说仍然属于重叠 I/O，它是到目前为止 Win32 平台下效率最高的多线程

网络编程模型，适用于具有高扩展性的高性能网络服务器的设计和实现。但是，需要注意的是，该模型仅能用于 Windows NT、Windows 2000 之后的操作系统。

18.2.5.1　基本函数

IOCP 模型的核心概念是完成端口，它是一种非常复杂的内核对象。在用户进程将一个或者（通常）多个套接口*与之绑定（Associate）后，当这些套接口上有重叠 I/O 操作请求完成时，系统就会自动将完成信息报放入一个 FIFO 类型的 I/O 完成队列（见图 18.5）中以达到通知应用程序的目的，而用户进程通常需要创建多个工作线程来处理这些通知信息。

IOCP 涉及到三个基本函数：CreateIoCompletionPort、GetQueuedCompletionStatus 和 PostQueuedCompletionStatus。第一个函数用于完成端口的创建和完成端口与套接口的绑定，第二个用于从完成队列中获取完成信息，最后一个用于模拟 I/O 请求的完成。

（1）CreateIoCompletionPort 函数定义如下：

```
    HANDLE CreateIoCompletionPort (
    HANDLE FileHandle,                  // handle to file
    HANDLE ExistingCompletionPort,      // handle to I/O completion port
    ULONG_PTR CompletionKey,            // completion key
    DWORD NumberOfConcurrentThreads     // number of threads to execute
                                        // concurrently
);
```

需要特别注意的是，该函数不仅用于完成端口的创建，还用于完成端口和套接口的绑定。因此它的使用可分为两个阶段：

阶段 1：完成端口的创建

我们只需要关心最后一个参数 NumberOfConcurrentThreads，其余三个参数可以依次设置为 INVALID_HANDLE_VALUE、NULL 和 0。NumberOfConcurrentThreads 定义了在该完成端口上可以同时工作的最大线程数，通常情况下该数目应与系统的处理器个数相同，通过将 NumberOfConcurrentThreads 赋值为 0 系统能自动完成这样的设置。可能的完成端口创建代码如下：

```
HANDLE hIOCP = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
```

如果函数操作成功返回创建的 IOCP 句柄，否则返回 NULL，这时可以用 GetLastError 函数来查询错误信息。

阶段 2：完成端口与套接口的绑定

这时四个参数的意义分别是：FileHandle，指向需要进行重叠 I/O 操作的设备句柄，在这儿就是套接口；ExistingCompletionPort，指向刚创建的完成端口的句柄；CompletionKey，完成键，又称单句柄（套接口）数据，用于指定 I/O 操作涉及的设备的特定信息，工作线程调用 GetQueuedCompletionStatus 函数可以获得该数据；NumberOfConcurrentThreads，设置为 0 即可。示例代码如下：

---

* 完成端口并不仅仅适用于套接口编程，它事实上是 Win32 平台下的一种通用 I/O 机制，可与之绑定的设备有文件、套接口、邮槽、管道等。

```
typedef struct _CPKey{
    SOCKET          svrSock;
} PER_SOCKET_DATA, *LPPER_SOCKET_DATA;
LPPER_SOCKET_DATA lpPerSocketData;
SOCKET svrSock = accept(sock, NULL, NULL);
CreateIoCompletionPort((HANDLE) svrSock, hIOCP, (DWORD) lpPerSocketData, 0);
```

为两个完全不同的功能提供同一个函数接口，非常容易引起混淆，因此程序设计人员通常用两个自定义函数来对 CreateIoCompletionPort 进行包装：

```
HANDLE CreateNewIoCompletionPort(DWORD dwNumberOfConcurrentThreads){
    return    (CreateIoCompletionPort(INVALID_HANDLE_VALUE,    NULL,    0,
    dwNumberOfConcurrentThreads));
}
BOOL AssociateWithIoCompletionPort(HANDLE hComPort, HANDLE hDevice, DWORD
    dwCompKey){
    return(CreateIoCompletionPort(hDevice,   hComPort,   dwCompKey,   0)   ==
    hComPort);
}
```

当然，这种复杂的设计也并非毫无意义，事实上可以用一次 CreateIoCompletionPort 函数调用就实现完成端口的创建和与一个设备相绑定的功能。下面的代码将一个服务套接口与新创建的完成端口 hIOCP 绑定，并且设置完成端口允许最多两个线程同时工作：

```
SOCKET svrSock = accept(sock, NULL, NULL);
HANDLE  hIOCP  =  CreateIoCompletionPort((HANDLE)svrSock,  NULL,  (DWORD)
    lpPerSocketData, 2);
```

（2）GetQueuedCompletionStatus 函数定义如下：

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,          // handle to completion port
    LPDWORD lpNumberOfBytes,        // bytes transferred
    PULONG_PTR lpCompletionKey,     // file completion key
    LPOVERLAPPED *lpOverlapped,     // buffer
    DWORD dwMilliseconds            // optional timeout value
);
```

该函数用于获取指定 I/O 完成端口的完成信息。如果没有完成信息，那么函数将阻塞直至 I/O 操作完成或者超时。五个参数中第一个和最后一个是输入参数，其余都是输出参数。CompletionPort，用于指定我们关心的完成端口的句柄；lpNumberOfBytes，用于返回 I/O 操作所传输的字节数；lpCompletionKey，返回 I/O 操作涉及的单文件句柄数据；lpOverlapped，返回进行重叠 I/O 操作时指定的重叠结构，使用 18.2.4 节介绍到的技术可以在 lpOverlapped 上追加任意数量的数据——称为单 I/O 操作数据；dwMilliseconds，指定函数等待完成信息出现的时间，以毫秒为单位，如果设置为 INFINITE 表示无限期等待，设置为 0 表示采取询问方式函数立即返回。

GetQueuedCompletionStatus 函数返回值的情况有些复杂，正确完整的示例代码如下：

```
BOOL  ret  =  GetQueuedCompletionStatus(hIOCP,  &dwNumBytes,  &CompKey,
    &pOverlapped, 1000);
if (ret) { // 非0返回值，成功
    // dequeues a completion packet for a successful I/O operation
}
else{ // 返回0，失败
    DWORD dwError = GetLastError();
    if (pOverlapped != NULL){
         // dequeues a completion packet for a failed I/O operation
         // dwError contains the reason for failure
    }
    else {
         // does not dequeue a completion packet
         // dwError contains the reason for failure
         if (dwError == WAIT_TIMEOUT) {
              // 超时
         }
         else {
              // Bad call to GetQueuedCompletionStatus
              // dwError contains the reason for the bad call
         }
    }
}
```

（3）PostQueuedCompletionStatus 函数定义如下：

```
BOOL PostQueuedCompletionStatus(
  HANDLE CompletionPort,                   // handle to an I/O completion port
  DWORD dwNumberOfBytesTransferred,        // bytes transferred
  ULONG_PTR dwCompletionKey,               // completion key
  LPOVERLAPPED lpOverlapped                // overlapped buffer
);
```

该函数用于向完成端口 CompletionPort 投递 I/O 完成信息包。三个参数 dwNumberOfBytesTransferred，dwCompletionKey 和 lpOverlapped 会在调用函数 GetQueuedCompletionStatus 时返回。

PostQueuedCompletionStatus 函数提供了与工作线程通信的手段。假设要终止应用进程，但是希望各个工作线程（假设有四个）能先进行相应的资源释放，那么 PostQueuedCompletionStatus 就是一个很好的途径。我们可以简单地以特定参数（一般将完成键置为 NULL）调用四次 PostQueuedCompletionStatus，当工作线程 GetQueuedCompletionStatus 返回该参数，也就获知了关闭通知，可以进行相应的线程退出处理了。

但是需要注意的是，如果并不是通知线程退出，这时线程会再次调用 GetQueuedCompletionStatus 函数，那么由于等待线程队列是 LIFO 方式的，很可能造成同一个线程接收了所有的 PostQueuedCompletionStatus 发送的信息，而其他线程无法接收的局面。要解决这

个问题，编程人员必须做好各个线程之间的同步工作。

18.2.5.2 重要概念

我们已经了解了完成端口涉及到的三个重要函数，在介绍模型的内部工作机制之前，先对几个重要概念进行分析和比较。

（1）单句柄数据（完成键）和单 I/O 操作数据

相对来说，单句柄数据对于套接口来说是静态数据，而单 I/O 操作数据是动态数据。一个完成端口上一般有多个套接口与之绑定，每个套接口对应一个单句柄数据结构（在完成端口和套接口绑定时设定），在同一个套接口上进行的每一次 I/O 操作都有一个单 I/O 操作数据结构与之对应，不同次之间的单 I/O 操作数据一般不同（在投递重叠 I/O 操作时进行设定）。我们以一个认证服务器为例来解释两者的区别和应用：假设服务器可以同时为多个客户端服务，认证过程需要多步网络数据交互。那么，当工作线程调用 GetQueuedCompletionStatus 后，它可以从单句柄数据（lpCompletionKey）获知是哪一个套接口（哪一个连接，也就是哪一个客户）上有 I/O 完成，从单 I/O 操作数据（lpOverlapped）获知这个连接上的协议交互已进行到哪一个阶段等。

（2）工作线程数与同时工作线程数

工作线程数与同时工作线程数是两个不同的概念，同时工作线程数一般与系统处理器的个数相同，而工作线程不仅包括了同时工作线程还包含了处于等待和暂停状态的工作线程。参照图 18.6，完成端口应用进程的工作进程是指等待线程队列、运行线程列表以及暂停线程列表中的所有线程的集合，而同时工作线程是指运行线程列表中的线程。

18.2.5.3 内部工作机制

当应用进程创建了一个完成端口，系统事实上为该完成端口创建了五个不同的数据结构，分别是设备列表、I/O 完成队列、等待线程队列、运行线程列表和暂停线程列表。

（1）设备列表

设备列表用于保存与该完成端口绑定的所有设备的信息，每条记录含设备句柄和完成键两个域。事实上完成键对于内核来说没有意义，它只是应用进程用于建立完成键和设备之间的对应关系，并以此来标志设备的一种手段。

当应用进程调用 CreateIoCompletionPort 函数将一个设备句柄与完成端口绑定，那么该设备的句柄以及应用进程指定的完成键信息将会被添加到设备列表中；而当一个设备被关闭，那么该设备所对应的表项就会被删除。



图 18.4 设备列表

（2）I/O 完成队列

I/O 完成队列用于保存 I/O 完成信息。当有 I/O 请求完成或者应用进程调用

PostQueuedCompletionStatus 函数，都会有相应的 I/O 完成信息加入此队列；当该队列不为空，那么某个线程的 GetQueuedCompletionStatus 函数调用将成功返回，同时被返回的完成信息会被从队列中删除，如图 18.5 所示。

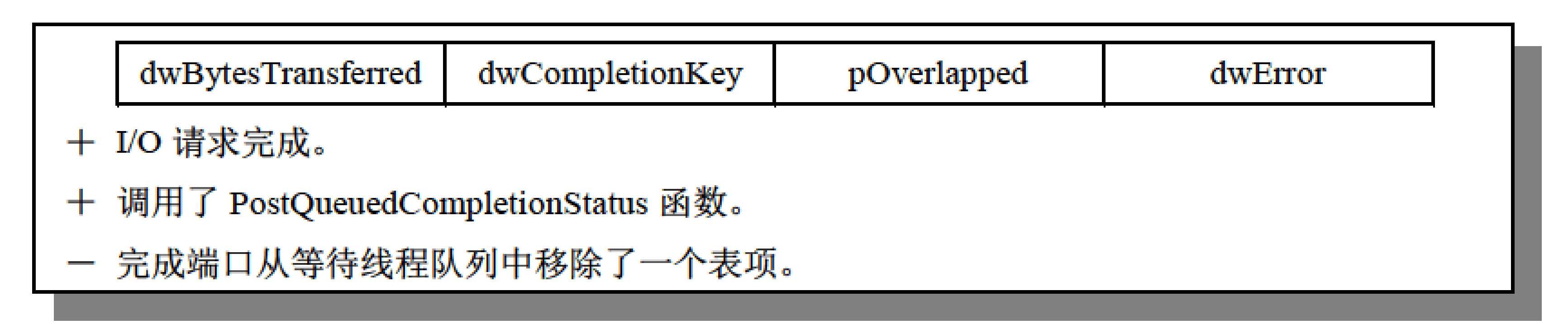


图 18.5　I/O 完成队列

（3）等待线程队列、运行线程列表和暂停线程列表

为了提高完成端口的多线程模型的效率，微软为它提供等待线程队列、运行线程列表和暂停线程列表这样三种数据结构，如图 18.6 所示，分别对应工作线程的三种状态，并在此基础上实现了非常精细的线程调度算法。

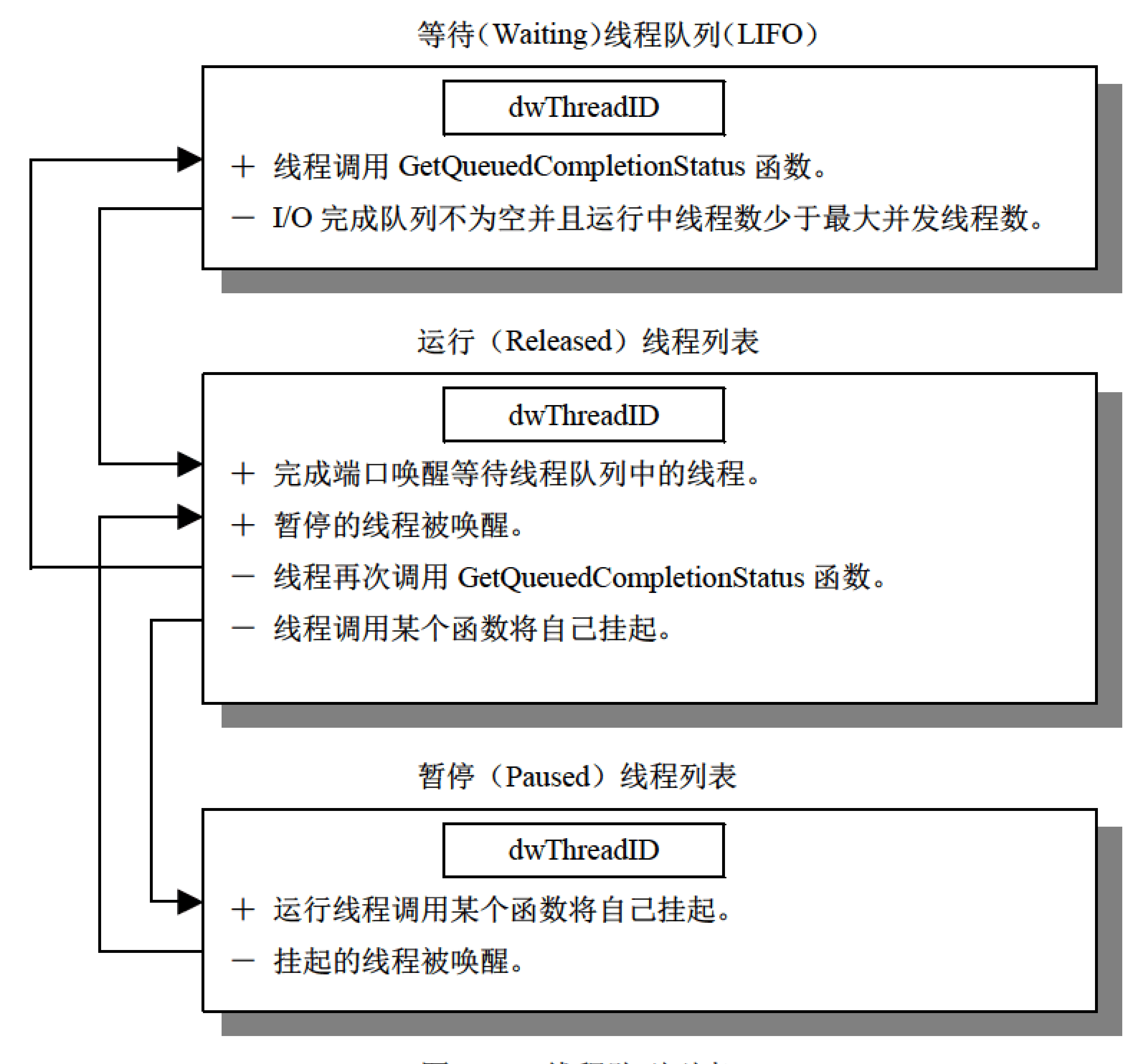


图 18.6　线程队列列表

线程初始时一般阻塞在 GetQueuedCompletionStatus 函数调用上，此时处于等待队列（LIFO）中；当有一个 I/O 完成信息包被投放至完成队列，系统会首先检查与完成端口联系的线程有多少正在运行中，如果运行线程数少于可同时工作线程数，那么等待线程队列中的最后一个到达的线程将被转为运行状态以处理该完成信息，此时该线程信息会被从等待线程



队列中删除并加入运行线程列表；否则，系统会让某一个运行线程处理完当前的完成信息后再处理该完成信息。这种调度机制的优点是减少了线程上下文环境的切换。

运行线程列表中的线程如果调用了 Sleep、WaitForSingleObject 等函数将自己挂起，那么系统会将该线程从运行线程列表中删除并转入暂停线程列表；此时如果完成队列中有完成信息待处理，那么系统将让等待线程队列中最后一个线程运行以处理该信息；考虑前面的线程 Sleep 结束并再次运行的情况，我们可以发现，在随后的一段时间里实际的同时工作线程数超过了 NumberOfConcurrentThreads 的规定值。当然这个阶段是非常短暂的，实际同时工作的线程数目会很快下降至不高于 NumberOfConcurrentThreads 值。这也就说明了为什么实际同时工作的线程数并不一定会少于或者等于 NumberOfConcurrentThreads 值。

18.2.5.4 两个问题

问题一：应用进程应该创建多少工作线程？

对于一般的服务器程序来说，建议创建 CPU 数目×2 这么多个工作线程，也就是说对于双 CPU 主机来说应该有四个工作线程。

但是在实际应用中，特别是对于一些性能要求很高的服务器来说，那么一个具有一定智能性的启发式算法是必需的。创建还是关闭工作线程，该算法必须综合考虑当前的线程总数、运行状态线程数、系统负荷等因素。相关内容可参阅《Programming Server-Side Applications for MS Windows 2000》一书。

问题二：如何投递 I/O 请求却又不引起完成队列中的信息变动？

当用完成端口模型进行套接口编程时，一个现实的问题是应用进程通常对数据的发送情况不关心，当调用 WSASend 函数时一般都认为发送会成功并立即返回，而事实上也确实如此。要进行发送操作而不用处理完成信息，可以将重叠结构的 hEvent 域置为一个合法的事件句柄与值 1 的“或”值，如下：

```
Overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
Overlapped.hEvent = (HANDLE) ((DWPRD_PTR) Overlapped.hEvent | 1);
// Do some I/O operation with Overlapped
```

当要关闭事件句柄时，必须首先将该句柄进行反处理：

```
CloseHandle((HANDLE) ((DWORD_PTR) Overlapped.hEvent & ~1));
```

我们已经讨论了完成端口的工作机制和相应的函数操作，对于具体的应用过程将在第 21 章给出详细的介绍。

第 19 章　套接口选项

套接口选项是套接口编程的重要内容，涉及到报文的广播和多播以及原始套接口等多个方面。本章第一节介绍套接口选项涉及的两个基本函数 setsockopt 和 getsockopt 以及相关的基本知识，并涉及到 WSAIoctl 函数的使用；在随后的三节分别介绍如何进行广播、多播和原始套接口编程。

19.1　套接口选项

使用 setsockopt 和 getsockopt 函数可以设置或者读取套接口的选项值，这两个函数的定义如下：

```
int setsockopt(SOCKET s, int level, int optname, const char FAR *optval, int
    optlen);
int getsockopt(SOCKET s, int level, int optname, char FAR *optval, int FAR
    *optlen);
```

其中参数 s[IN/IN]（分别对应 setsockopt 和 getsockopt，下同）是我们感兴趣的套接口描述字，level[IN/IN]是套接口选项的级别，optname[IN/IN]是用户指定的选项名称，optval[IN/OUT]是一个指向变量的指针，它的大小由 optlen[IN/INOUT]指定，通过 optval 可以设置或者读取指定选项的值。

如果没有错误发生，函数返回 0；否则返回 SOCKET_ERROR。

Winsock 的套接口选项主要有两种类型：布尔型，用于启用或禁止套接口的某个属性；选项参数，整型或者结构体，设置特定的值。为了启用某个布尔型选项，optval 应指向某个非零整数，若要禁用则指向一个零值整数，同时 optlen 参数值应该等于布尔型大小，即 sizeof(BOOL)；对于其他选项类型，optval 应指向整型或者结构体，optlen 值为整型或者结构体的大小。

Winsock 支持的全部选项级别有 IPPROTO_IP、IPPROTO_IPV6、IPPROTO_RM、IPPROTO_TCP、IPPROTO_UDP、NSPROTO_IPX、SOL_APPLETALK、SOL_IRLMP 和 SOL_SOCKET。下面以选项的设置为主介绍其中的几个级别的常用选项参数。

19.1.1　SOL_SOCKET

（1）SO_BROADCAST，BOTH，布尔型（BOOL）。

用于允许/禁止发送广播报文。该选项将在 19.2 节详细讨论。

（2）SO_LINGER，BOTH，LINGER 结构型，定义如下：

```
struct linger {
  u_short    l_onoff;
  u_short    l_linger;
}
```

其中 l_onoff 用于指定如果调用 closesocket 函数后还有数据待发送，是否需要延时关闭套接口。l_linger 用于设定启用 SO_LINGER 时等待数据发送的时间。

该选项用于控制当关闭套接口时如果缓冲区里仍有数据待发送，closesocket 函数的处理方式：要不要延缓关闭以等待数据发送完毕。

要启用 SO_LINGER，应用程序应该给 l_onoff 赋一个非零值，并且设置 l_linger 为 0 或者需要的等待时间（以秒为单位），然后调用 setsockopt 函数；为了指定 SO_DONTLINGER（禁用 SO_LINGER），l_onoff 应该被设置为 0。需要注意的是，应该避免在非阻塞套接口上启用 SO_LINGER 选项同时设置非 0 的延时值。

（3）SO_DONTLINGER，BOTH，布尔型，用于禁用 LINGER。

设置该选项后，如果有数据待发送那么套接口关闭的操作将被延缓。该选项等价于以 l_onoff 为 0 设置 SO_LINGER 选项。

（4）SO_KEEPALIVE，BOTH，布尔型，周期性地测试连接是否存活。

应用进程为一个 TCP 套接口设置了 SO_KEEPALIVE 选项后，如果固定时间内（默认为 2 个小时）在此套接口上通信的两个方向上都没有数据流量，那么系统会自动向对方发送一个保持存活探测分组以检测对方的状态（正常、服务关闭或者无响应）。如果探测认为连接已断开，那么在此套接口上的任何操作都会返回错误码 WSAENETRESET，随后的操作返回 WSAENOTCONN。

在很多情况下，两个连接的套接口之间会长时间没有数据发送，这就意味着在长期存活的进程中空闲的套接口可能几分钟、几小时、甚至于几天都不会提交数据，也就不会发现对方可能已断线。因此，假设某个客户端崩溃了，那么另外一端主机的资源，例如 CPU 时间和内存，就会浪费在那个永远不会响应的客户端上。

很自然地，我们会希望 SO_KEEPALIVE 选项能用于连接存活检测的心跳测试，以避免这种情况的发生。由于系统的默认设置是空闲 2 小时才发送一个探测分组，并不能保证检测的实时性，因此必须修改时间间隔参数。在 Windows 2000 之前，对时间间隔参数的修改都是操作系统级别的，也就是说这种修改会影响到所有打开此选项的套接口。在 Windows 2000 及其之后的操作系统中，对 I/O 控制函数 WSAIoctl 引入了一个新的控制码 SIO_KEEPALIVE_VALS（在 MSTcpIP.h 中定义，在该文件中还定义了其他一些新的 WSAIoctl 选项如 SIO_RCVALL），可用它针对单个套接口更改其“保持活动”值以及发送的间隔时间。该控制码操作涉及到的 tcp_keepalive 结构体定义如下：

```
struct tcp_keepalive {
    u_long  onoff;
    u_long  keepalivetime;
    u_long  keepaliveinterval;
};
```

但是，我们仍然建议不采用 SO_KEEPALIVE 选项而是自己编写应用层面上的超时检测机制（心跳检测）来保证连接的存活。最常见的方法是使用带外数据，这样也能确保不会干扰正常的数据通信。

（5）SO_RCVBUF，BOTH，int 型，为每个套接口设定/读取接收缓冲区的大小。

该缓冲区的大小与 SO_MAX_MSG_SIZE 常量值或者 TCP 窗口大小有关。应用进程可以为不同的套接口申请不同的接收缓冲区大小，但是当 setsockopt 成功时并不说明内核就提供了请求大小数量的缓冲区，可以用同样的选项调用 getsockopt 来检测实际提供的缓冲区大小值。

如果将 SO_RCVBUF 设置为 0，将导致接收数据在比 Winsock 更底层被缓存，因此并不能提高反而会影响系统的性能。

（6）SO_SNDBUF，BOTH，int 型，为每个套接口设定/读取发送缓冲区的大小。

该缓冲区的大小与 SO_MAX_MSG_SIZE 常量值或者 TCP 窗口大小有关。应用进程可以为不同的套接口申请不同的发送缓冲区大小，但是当 setsockopt 成功时并不说明内核就提供了请求大小数量的缓冲区，我们可以用同样的选项调用 getsockopt 来检测实际提供的缓冲区大小值。

如果应用进程将 SO_SNDBUF 设置为 0 并且调用阻塞数据发送，那么内核就会将应用进程的缓存锁定直至发送 API 调用成功。仅当我们需要开发一个高性能的服务器程序时，才考虑将套接口的发送缓冲区设置为 0。需要注意的是，这时的服务器程序应该是能同时发送多个重叠 send 的，而不需要线性地等待一个发送操作完毕再进行下一个发送函数调用。

（7）SO_REUSEADDR，BOTH，布尔型，允许套接口绑定一个已在使用的地址。

默认情况时，套接口是无法绑定一个已在使用的地址的。我们知道，不同的连接是由双方的<地址，端口>来区分的，因此连接不同的目的端时两个进程使用本地的同一个端口也是可行的。在重用本地地址时，必须启用 SO_REUSEADDR 选项。

事实上，SO_REUSEADDR 选项更多的是在服务器编程时需要使用。当服务器进程需要重启时，经常会碰到监听端口尚未完全关闭的情况（处于 TIME_WAIT 状态），这时如果没有启用 SO_REUSEADDR 选项，那么 bind 操作就会报 WSAEADDRINUSE 错误。因此，一般进行服务器编程时都会对监听端口启用 SO_REUSEADDR。

（8）SO_RCVTIMEO，BOTH，struct timeval 结构型，用于设置数据接收超时值。

该选项用于在一个阻塞套接口为接收函数设定一个超时值。当调用接收函数时，如果在 SO_RCVTIMEO 指定的时间内没有数据到来，那么函数调用也会结束并且返回错误 10060（WSAETIMEDOUT）。

（9）SO_SNDTIMEO，BOTH，struct timeval 结构型，用于设置数据发送超时值。

该选项用于在一个阻塞套接口为发送函数设定一个超时值。当调用发送函数时，如果在 SO_RCVTIMEO 指定的时间内数据还未发送成功，那么函数调用也会结束并且返回错误 10060（WSAETIMEDOUT）。

以 SO_RCVTIMEO 为例说明 SOL_SOCKET 选项的使用。下面的例子演示了如何为 TCP/UDP 套接口设置并读取接收超时值。对于 TCP 套接口，程序连接 WEB 服务器 202.119.24.32（www.seu.edu.cn），并希望能在 10 秒内收到对方主动发送来的数据；对于 UDP 套接口，程序监听本地的 9999 端口然后等待外来的 UDP 报文。

```
**************************** 程序19.1 SO_RCVTIMEO ************************
#pragma comment(lib, "ws2_32.lib")
#include <STDIO.H>
#include <WINSOCK2.H>

#define TCP

void HandleError(char *);

int main(int argc, char* argv[])
{
    WSAData wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);
    DWORD begin, finish;

    struct timeval tv;
    tv.tv_sec = 10000;
    tv.tv_usec = 0;
    int optlen = sizeof(tv);

#ifdef TCP
    SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    printf("TCP Test!\n");

    struct sockaddr_in to;
    int len = sizeof(to);
    memset(&to, 0, len);
    to.sin_addr.s_addr = inet_addr("202.119.24.32");
    to.sin_family = AF_INET;
    to.sin_port = htons(80);

    if(connect(sock, (struct sockaddr *) &to, len) == SOCKET_ERROR){
        HandleError("connect");
        closesocket(sock);
        WSACleanup();
        return -1;
    }

    if(setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (char *) &tv, optlen) ==
    SOCKET_ERROR){
        HandleError("setsockopt");
        closesocket(sock);
        WSACleanup();
        return -1;
    }

    char buf[100];
    begin = GetTickCount();
```

```
    if(recv(sock, buf, 100, 0) == SOCKET_ERROR){
        HandleError("recv");
        if(WSAGetLastError() == WSAETIMEDOUT){
            printf("WSAETIMEDOUT\n");
        }
    }
    finish = GetTickCount();

#else // NOT TCP!

    SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
    printf("UDP Test!\n");

    struct sockaddr_in local;
    int len = sizeof(local);
    memset(&local, 0, len);
    local.sin_addr.s_addr = INADDR_ANY;
    local.sin_family = AF_INET;
    local.sin_port = htons(9999);

    if(bind(sock, (struct sockaddr *) &local, len) == SOCKET_ERROR){
        HandleError("bind");
        closesocket(sock);
        WSACleanup();
        return -1;
    }

    if(setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (char *) &tv, optlen) ==
    SOCKET_ERROR){
        HandleError("setsockopt");
        closesocket(sock);
        WSACleanup();
        return -1;
    }

    char buf[100];
    begin = GetTickCount();
    if(recvfrom(sock, buf, 100, 0, NULL, NULL) == SOCKET_ERROR){
        HandleError("recvfrom");
        if(WSAGetLastError() == WSAETIMEDOUT){
            printf("WSAETIMEDOUT\n");
        }
    }
    finish = GetTickCount();
#endif

    printf("实际等待时间(毫秒): %d\n", finish - begin);
```

```
    memset(&tv, 0, optlen);
    if(getsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (char *) &tv, &optlen) ==
    SOCKET_ERROR){
        HandleError("getsockopt");
    }
    printf("设置等待时间(毫秒): %d\n", tv.tv_sec);

    closesocket(sock);
    WSACleanup();

    return 0;
}

void HandleError(char *func)
{
    int errCode = WSAGetLastError();

    char info[65] = {0};
    _snprintf(info, 64, "%s:        %d\n", func, errCode);
    printf(info);
}
*************************************************************************
```

程序的输出结果分别为：

```
TCP Test!                             UDP Test!
recv:              10060              recvfrom:          10060
WSAETIMEDOUT                          WSAETIMEDOUT
实际等待时间(毫秒): 10516              实际等待时间(毫秒): 10532
设置等待时间(毫秒): 10000              设置等待时间(毫秒): 10000
```

19.1.2 IPPROTO_IP

（1）IP_HDRINCL，BOTH，布尔型，仅适用于原始套接口（SOCK_RAW）。

如果应用程序希望能接收 IP 层及 IP 层以上的所有数据或者自行组装包含 IP 层在内的报文，那么可以设置该选项为 TRUE。将在 19.4 节对该选项进行详细描述。

（2）IP_ADD_MEMBERSHIP，SET，struc ip_mreq 结构型，用于加入多播组。

该选项用于将指定网络接口上的套接口加入 IP 多播组，此套接口必须是 AF_INET 地址族并且类型为 SOCK_DGRAM。其中 struct ip_mreq 结构定义如下：

```
struct ip_mreq {
    struct in_addr imr_multiaddr;   /* IP multicast address of group */
    struct in_addr imr_interface;   /* local IP address of interface */
};
```

其中 imr_multiaddr 对应于打算加入的多播组的 IPv4 地址；而 imr_interface 是本地接口的 IP 地址，也可以设置为 INADDR_ANY，表明选择的是默认接口。

（3）IP_DROP_MEMBERSHIP，SET，struc ip_mreq 结构型，用于退出多播组。该选项的使用与 IP_ADD_MEMBERSHIP 类似。

（4）IP_MULTICAST_IF，BOTH，DWORD 类型，设置/读取多播的本地接口。

IP_MULTICAST_IF 选项用于设置或读取本地接口。在设置了本地的多播外出接口后，本地机器以后发出的任何多播数据都会经由它传送出去，该选项适用于多穴主机。optval 参数是一个无符号的长整数值，对应于本地接口的 IPv4 地址。可用 inet_addr 函数将一个字串形式的 IP 地址（点分十进制）转换成一个无符号的长整数值。

（5）IP_MULTICAST_LOOP，BOTH，布尔型，用于启用或者禁止多播报文环回。

在默认的情况下，当发送 IP 多播数据时，如果发送套接口本身也属于该多播组，那么数据会原封不动地返回一份至套接口——环回（loopback）。若将该选项设为 FALSE，发出的任何数据都不会投递至套接字的进入数据队列中。

（6）IP_MULTICAST_TTL，BOTH，DWORD 类型，设置/读取套接口上 IP 多播的 TTL 值。

在默认情况下，多播数据报采用的 TTL 值为 1。IP_MULTICAST_TTL 选项可用于读取或者设定该值。多播 TTL 值的大小影响到多播数据的传播范围，只有在有效范围内的组成员才会收到数据。

19.2　广　　播

我们知道，在 IPv4 中报文的传输分为三种方式：单播、广播和多播。对于 TCP 来说，一个进程通过连接与另外一个进程进行通信，所有的报文交互的都是单播方式的，不存在广播和多播的概念。因此，对于用户进程来说，只有通过 UDP 套接口才能实现广播和多播。

19.2.1　报文的发送

在默认情况下，UDP 套接口是无法发送广播报文的，在 19.1.1 节介绍过套接口启用/禁止广播是通过 SOL_SOCKET→SO_BROADCAST 选项来完成的。通过下面的代码段，可以确认广播选项是默认关闭的。

```
BOOL bBroadcast;
int optlen = sizeof(bBroadcast);
if(getsockopt(sock, SOL_SOCKET, SO_BROADCAST, (char *) & bBroadcast, &optlen)
    == SOCKET_ERROR){
        HandleError("getsockopt");
        closesocket(sock);
        WSACleanup();
        return -1;
    }
if(bBroadcast)
    printf("Broadcast enabled default!\n");
else
```

```
    printf("Broadcast disabled default!\n");
```

程序输出“Broadcast disabled default!”。因此，要发送广播报文，必须首先启用 SO_BROADCAST 选项，如下：

```
bBroadcast = true;
optlen = sizeof(bBroadcast);
if(setsockopt(sock, SOL_SOCKET, SO_BROADCAST, (char *) &bBroadcast, optlen)
    == SOCKET_ERROR){
    HandleError("setsockopt");
    closesocket(sock);
    WSACleanup();
    return -1;
}
```

此时仍然可以调用 getsockopt 函数检查是否设置成功，在成功地启用了 SO_BROADCAST 后，就可以发送广播报文了（否则 sendto 函数会返回 10013 错误，即 WSAEACCES）：

```
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_addr.s_addr = INADDR_BROADCAST;// inet_addr("202.119.9.255");
addr.sin_family = AF_INET;
addr.sin_port = htons(9999);

const char *msg = "Hello! Broadcast Test!";
int len = strlen(msg);

if(sendto(sock, msg, len, 0, (struct sockaddr *) &addr, sizeof(addr)) ==
    SOCKET_ERROR){
    HandleError("sendto");
    closesocket(sock);
    WSACleanup();
    return -1;
}
```

其中 INADDR_BROADCAST 是 Winsock 定义的常量 0xffffffff，对应于受限广播地址 255.255.255.255。在测试时，我们发现一个奇怪的现象，当不使用受限广播地址，而是使用指向网络的广播地址 202.119.9.255 时（见代码斜体部分），无论 SO_BROADCAST 选项处于启用还是禁止状态，报文均能被广播出去。

另外一个问题是，在发送广播报文时需要注意 UDP 数据报的大小限制。源自 Berkeley 的内核不允许广播数据报被分片，但是在笔者的 Windows 2000 系统中不存在这个问题。为了测试 Windows 2000 对大 UDP 报文的处理，把 msg 定义为一个 2000 个字节的数组、len 赋值为 2000，调用 sendto 函数后，同在 202.119.9 网段的另一台主机上的网络监视器返回的结果如图 19.1 和图 19.2 所示。

可以看到，事实上接收端收到两个报文：1472 字节应用数据的 UDP 报文和 528 个字节的 IP 分片报文。而在运行下一节提到的接收程序时，应用进程能直接收到重组后的 2000 字



节 UDP 数据（recvfrom 返回数值 2000）。

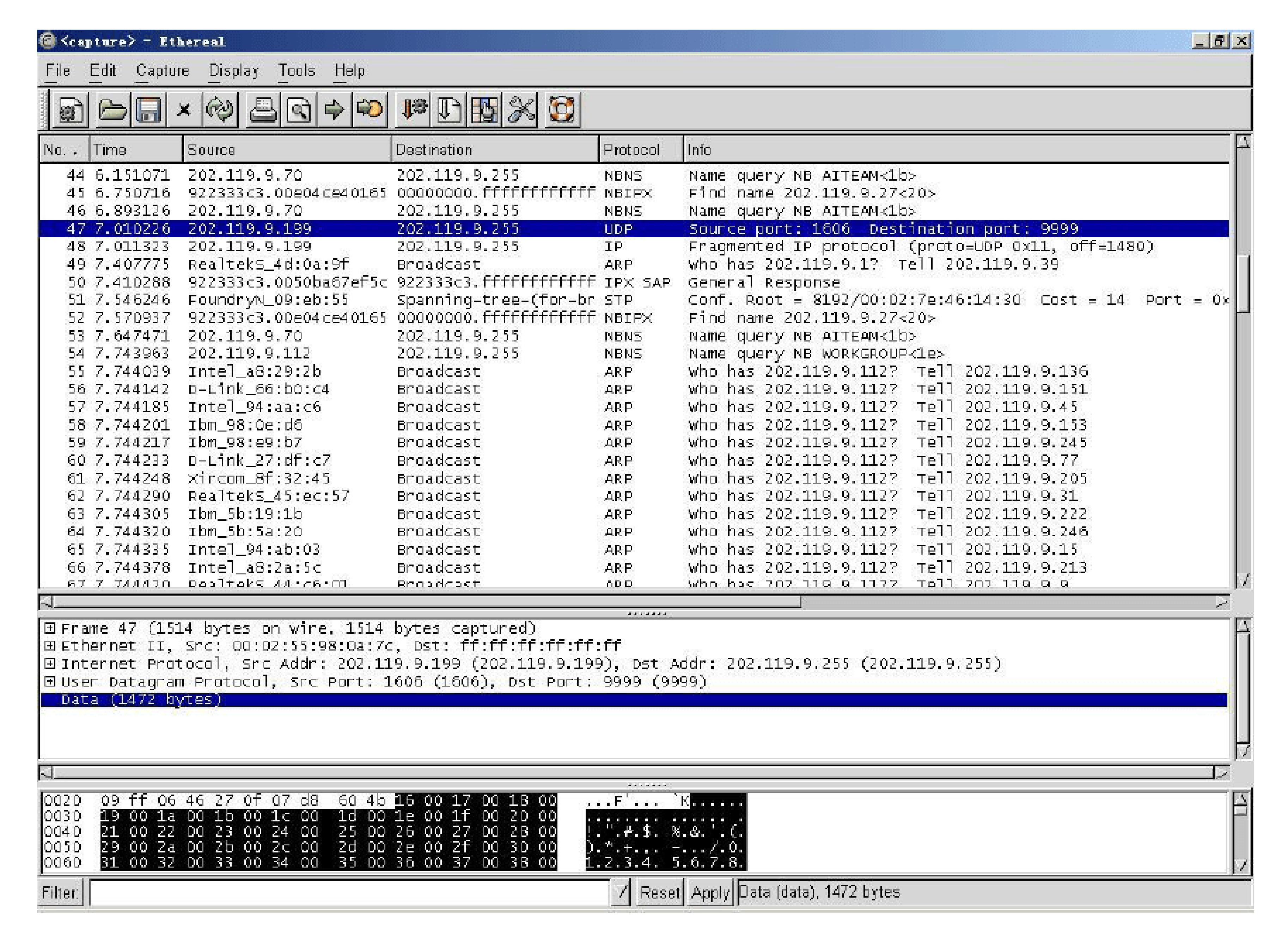


图 19.1　UDP 广播报文 A

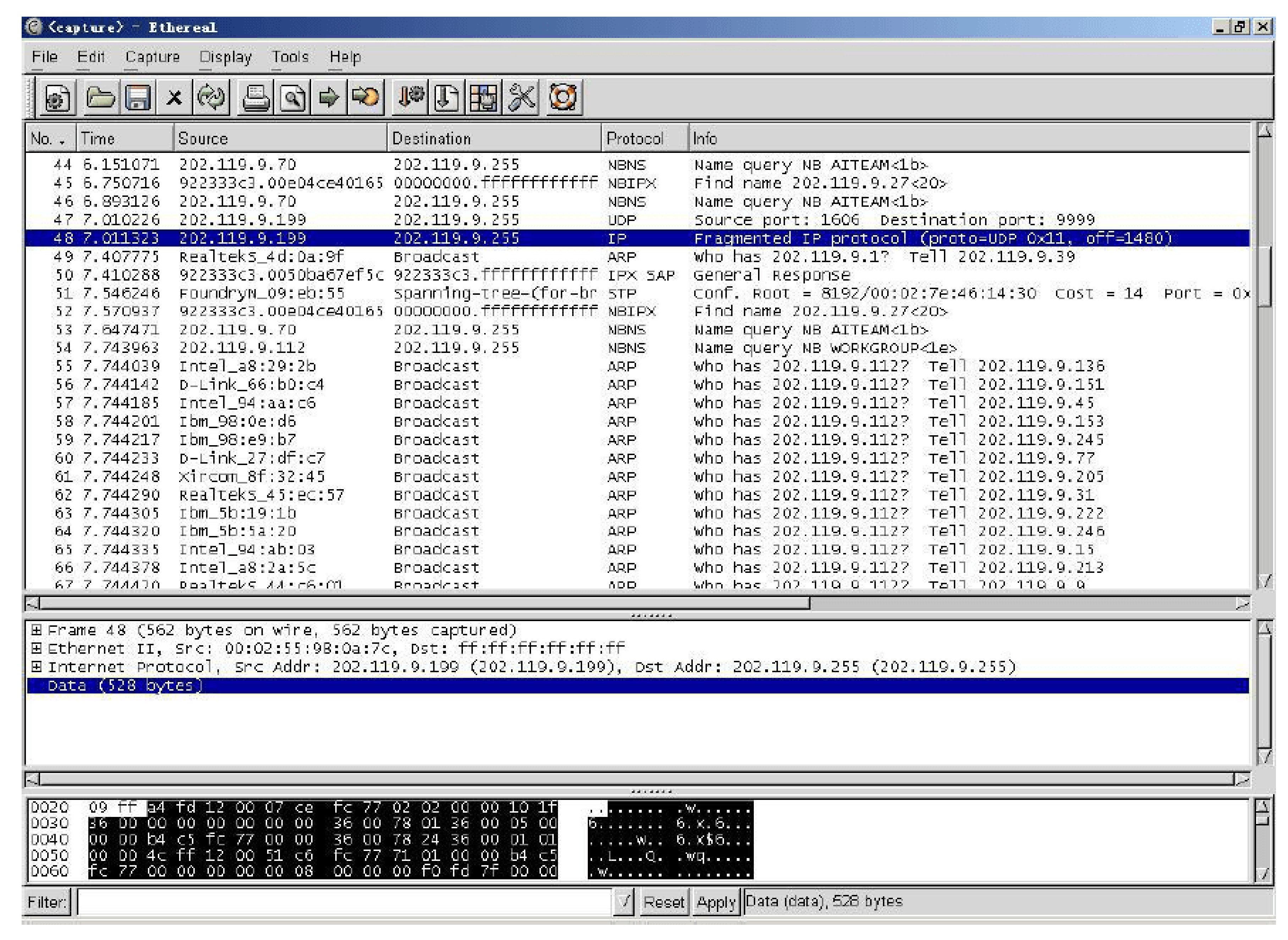


图 19.2　UDP 广播报文 B



由此可以得出结论，Windows 2000 系统允许广播数据报分片，因此广播报文可以大于外出接口的 MTU。但是，如果需要考虑程序的可移植性，广播应用系统应该将报文限制在 1472 字节以内。

19.2.2　广播报文的接收

接收广播报文不需要任何额外的设置，创建 UDP 套接口后，绑定所需的端口即可，以上面的发送程序为例，接收方程序段如下：

```
SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(9999);
addr.sin_addr.s_addr = INADDR_ANY;
bind(sock, (struct sockaddr *) &addr, sizeof(addr));

int ret;
char buf[5000];
while(1){
    ret = recvfrom(sock, buf, 5000, 0, NULL, NULL);
    if(ret  == SOCKET_ERROR){
        printf("recvfrom: %d\n", WSAGetLastError());
        break;
    }
    else
        printf("recvd %d bytes\n", ret);
}
```

19.3　多　　播

设计报文广播方式的最初的目的是用于资源发现和减少数据交互量。但事实上，由于报文广播时，同一网段内的所有主机，无论有没有参与广播应用，都必须完成对数据报的处理。被广播的 UDP 报文会被接收主机的系统协议栈逐层处理，直到传输层将其交付监听相应端口的应用进程或者丢弃。因此，频繁的大数据量的报文广播会严重影响网络上的其他主机的正常运行。而多播方式在具有广播的优点的同时，很好地解决了这个问题。

多播编程涉及到 19.1.2 节中除 IP_HDRINCL 之外的所有选项，我们已经介绍了这些选项的基本功能，下面直接给出应用这些选项的简单的 lib 库。*

* 出于与伯克利套接口的兼容性考虑，此处对多播的介绍基于 Winsock 1 版本，不涉及 Winsock 2 版本中的相关函数。

19.3.1　一个简单的多播库

多播库头文件如下：

```
******************************** 程序19.2.A ********************************
#ifndef _MCASTLIB_H_
#define _MCASTLIB_H_    1

#include <WinSock2.h>
#include <WS2tcpip.h>

#ifdef __cplusplus
extern "C" {
#endif
int mc_join(SOCKET s, struct in_addr *mcaddr, struct in_addr *local_if);
int mc_setIF(SOCKET s, const DWORD local_out_if);
int mc_getIF(SOCKET s, DWORD *local_out_if);
int mc_setTTL(SOCKET s, const DWORD ttl);
int mc_getTTL(SOCKET s, DWORD *ttl);
int mc_setLoop(SOCKET s, const BOOL flag);
int mc_getLoop(SOCKET s, BOOL *flag);
int mc_leave(SOCKET s, struct in_addr *mcaddr, struct in_addr *local_if);
#ifdef __cplusplus
}
#endif
#endif
****************************************************************************
```

多播库的具体实现如下：

```
******************************** 程序19.2.B ********************************
#include "MCastLib.h"
// 本地接口local_if加入多播组mcaddr
int mc_join(SOCKET s, struct in_addr *mcaddr, struct in_addr *local_if)
{
    struct ip_mreq  mreq;
    memcpy(&(mreq.imr_interface), local_if, sizeof(struct in_addr));// local
    if
    memcpy(&(mreq.imr_multiaddr),   mcaddr,   sizeof(struct   in_addr));//
    mutilcast group address

    return(setsockopt(s,  IPPROTO_IP,  IP_ADD_MEMBERSHIP,  (char  *)  &mreq,
    sizeof(mreq)));
}

// 为多播报文设置外出接口
int mc_setIF(SOCKET s, const DWORD local_out_if)
```

```
{
    return(setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *) &local_out_if,
    sizeof(local_out_if)));
}

// 获取多播报文的外出接口
int mc_getIF(SOCKET s, DWORD *local_out_if)
{
    int len = sizeof(DWORD);
    return(getsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *) local_out_if,
    &len));
}

// 设置外出多播报文的ttl值，默认为1
int mc_setTTL(SOCKET s, const DWORD ttl)
{
    return(setsockopt(s,  IPPROTO_IP,  IP_MULTICAST_TTL,  (char  *)  &ttl,
    sizeof(ttl)));
}

// 获取外出多播报文的ttl 值
int mc_getTTL(SOCKET s, DWORD *ttl)
{
    int len = sizeof(DWORD);
    return(getsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, (char *) ttl, &len));
}

// 启用或者禁止多播报文环回
int mc_setLoop(SOCKET s, const BOOL flag)
{
    return(setsockopt(s,  IPPROTO_IP,  IP_MULTICAST_LOOP,  (char  *)  &flag,
    sizeof(flag)));
}

// 获取本地多播环回状态
int mc_getLoop(SOCKET s, BOOL *flag)
{
    int len = sizeof(BOOL);
    return(getsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, (char *) flag, &len));
}

// 本地接口local_if离开多播组mcaddr
int mc_leave(SOCKET s, struct in_addr *mcaddr, struct in_addr *local_if)
{
    struct ip_mreq  mreq;
    memcpy(&(mreq.imr_interface), local_if, sizeof(struct in_addr));// local
    if
    memcpy(&(mreq.imr_multiaddr),   mcaddr,   sizeof(struct   in_addr));//
```

```
    mutilcast group address

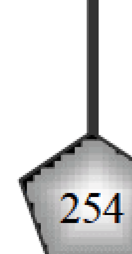
    return(setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP, (char *) &mreq,
    sizeof(mreq)));
}
*************************************************************************
```

19.3.2　接收多播数据

加入多播组并接收多播数据，我们所要做的工作可能比大部分人想象的都要简单：创建一个 UDP 套接口，绑定本地端口，调用上一节中的 mc_join 函数加入相应的多播组，然后就可以接收数据了。加入多播组是要告诉本机的 IP 层和数据链路层接收发送到这个组的多播数据；捆绑端口则是向 UDP 层通告该应用程序希望接收发送到此端口的数据报。一个简单的例子如下：

```
struct sockaddr_in local;
memset(&local, 0, sizeof(local));
local.sin_family = AF_INET;
local.sin_port = htons(9999);
local.sin_addr.s_addr = inet_addr("202.119.9.199");
// 绑定<202.119.9.199, 9999>
if(bind(sock, (struct sockaddr *) &local, sizeof(local)) == SOCKET_ERROR){
    printf("bind: %d\n", WSAGetLastError());
}
// 202.119.9.199加入多播组226.1.2.3, <226.1.2.3, 9999>
struct in_addr mcaddr;
mcaddr.s_addr = inet_addr("226.1.2.3");
if(mc_join(sock, &mcaddr, &(local.sin_addr)) == SOCKET_ERROR){
    printf("Join Multicast Group: %d\n", WSAGetLastError());
}
// 接收数据，考虑此时能收到发往哪些目的地址的UDP报文
char buf[65];
while(1){
    memset(buf, 0, 65);
    if(recvfrom(sock, buf, 64, 0, NULL, NULL) == SOCKET_ERROR){
        printf("recvfrom: %d", WSAGetLastError());
        break;
    }
    else
        printf("recvd: %s\n", buf);
}
```

上述的代码使套接口 sock 加入多播组 226.1.2.3，并接收发往该组的数据。打开网络监视器会发现，当调用 mc_join 函数加入多播组时，内核会自动向该组发送一个“IGMP v2 Memebership Report”报文，该报文会被组内的所有主机以及路由器接收。这也就是系统在后台为该调用所作的工作之一。如图 19.3 所示。



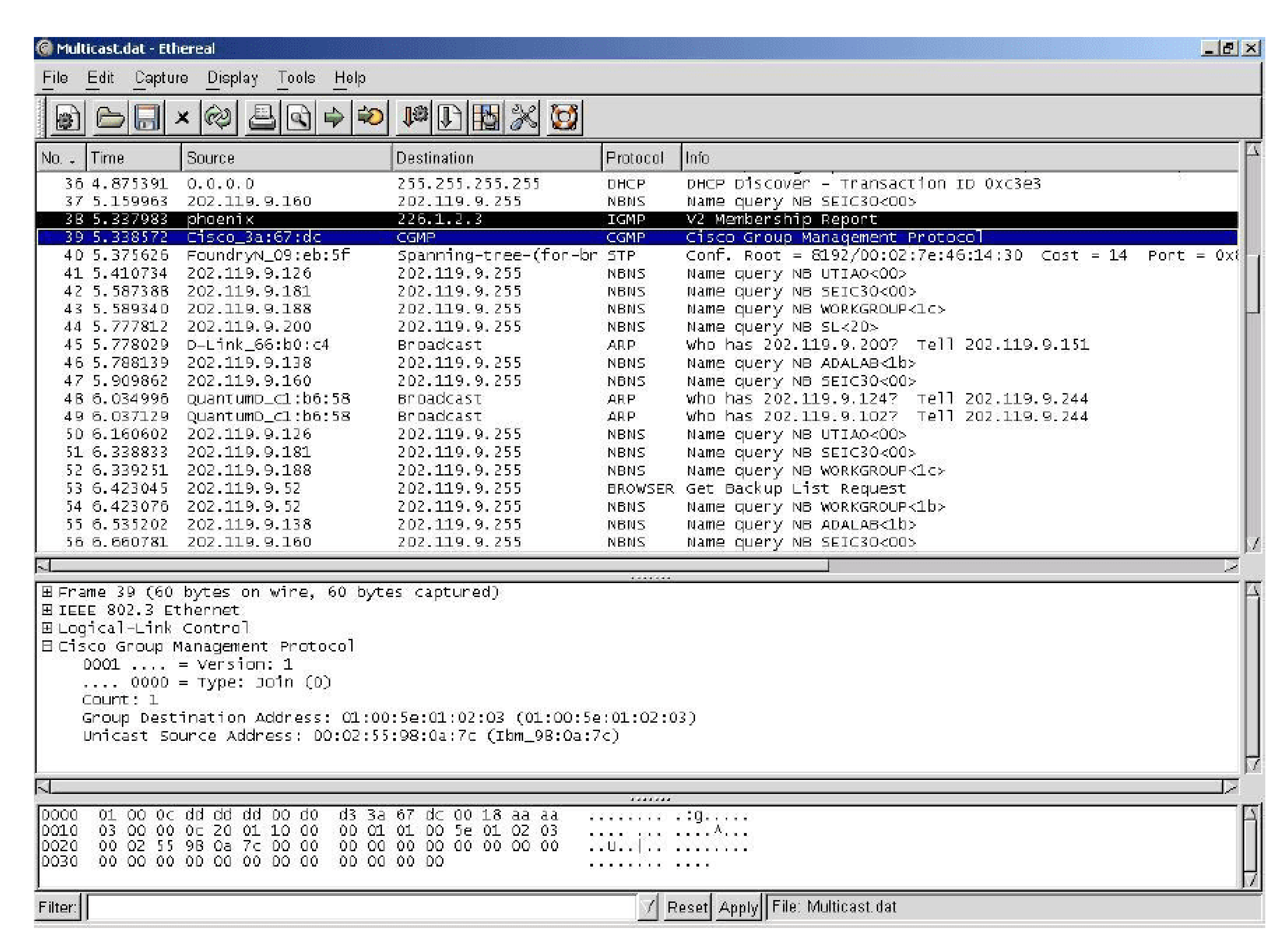


图 19.3 加入多播组

在加入了多播组之后，就可以像一般的 UDP 应用程序那样来进行报文的接收工作了。但是，可以想象，在执行了上述代码加入了 226.1.2.3 多播组后，程序不仅会接收到所有的发送到<226.1.2.3, 9999>地址的报文，还会收到发往<202.119.9.199, 9999>的报文。在某些 UNIX 系统中，可以采用将套接口绑定到多播系统所使用的多播地址上的方法来使内核自动将非多播报文过滤掉。但是在 Winsock 中，这种地址绑定是不允许的（bind 会报 10049 错误，WSAEADDRNOTAVAIL），必须根据报文的源地址或者应用层数据来进行过滤。

在套接口关闭时，无论有没有显式地调用 mc_leave 函数，进程都会离开多播组，如图 19.4 所示。如果此时本机没有其他进程加入了该组，那么内核将向 224.0.0.2（所有路由器组）发送“IGMP v2 Leave Group”报文。

19.3.3 发送多播数据

如果不关心多播出口、TTL 值和多播环回的问题，那么向一个多播组发送数据是非常简单的，只需要像普通的 UDP 报文发送那样调用 sendto 函数，惟一的差别是目标地址是多播组地址。

但是，如果需要对多播属性进行更改，如 TTL 值（TTL 值的设置对大部分多播应用系统来说是必需的），那么必须在调用 sendto 函数之前完成相应的设置。如表 19.1 所示。

下面的程序段读取了多播报文的默认 TTL 值和环回状态，在将 TTL 值改为 219 后，向多播组 226.1.2.3 发送了数据“Hello！”。

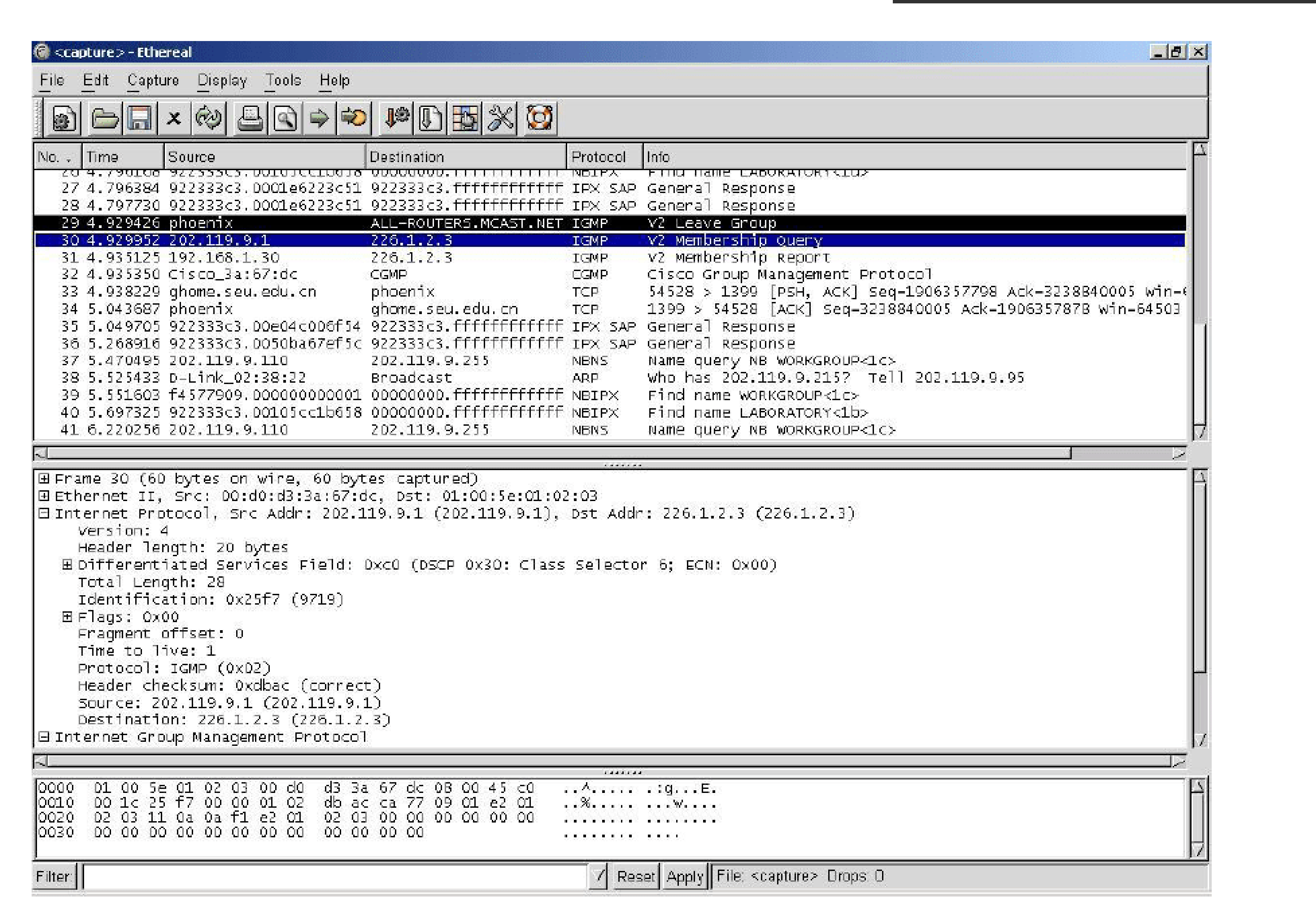


图 19.4　离开多播组

表19.1　多播TTL值

TTL 阈值	描　　述
0	Restricted to the same host
1	Restricted to the same subnet
32	Restricted to the same site
64	Restricted to the same region
128	Restricted to the same continent
255	Unrestricted in scope

```
SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
// 获取默认的多播报文TTL值和环回状态
DWORD ttl;
if(mc_getTTL(sock, &ttl) == SOCKET_ERROR){
    printf("mc_getTTL: %d\n", WSAGetLastError());
}
BOOL loop;
if(mc_getLoop(sock, &loop) == SOCKET_ERROR){
    printf("mc_getLoop: %d\n", WSAGetLastError());
}
printf("Multicast default: TTL = %d, LoopBack = %d\n", ttl, loop);
// 设置多播TTL值为219
ttl = 219;
if(mc_setTTL(sock, ttl) == SOCKET_ERROR){
```

```
    printf("mc_getTTL: %d\n", WSAGetLastError());
}
// 向多播组226.1.2.3发送数据
struct sockaddr_in to;
memset(&to, 0, sizeof(to));
to.sin_addr.S_un.S_addr = inet_addr("226.1.2.3");
to.sin_family = AF_INET;
to.sin_port = htons(9999);
char *buf = "Hello!";
int res = sendto(sock, buf, 6, 0, (struct sockaddr *) &to, sizeof(to));
if(res == SOCKET_ERROR){
    HandleError("sendto");
}
else
    printf("Send out %d bytes!\n", res);
```

由图 19.5 可以发现，程序成功地向多播组 226.1.2.3 发送了数据“Hello！”，并且多播报文的 TTL 值已被设置为 219。

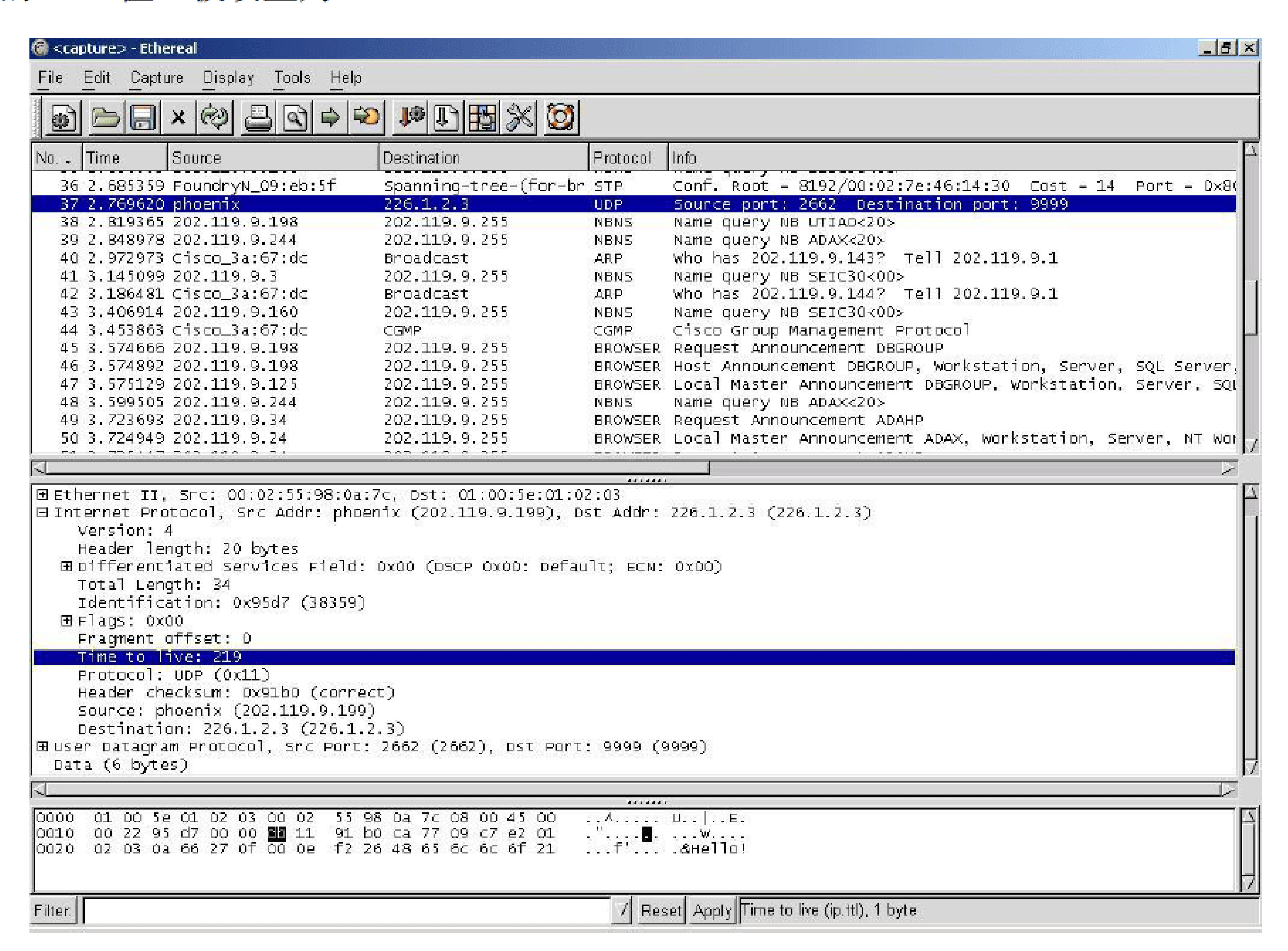


图 19.5　发送多播报文

19.4　原始套接口编程

原始套接口（SOCK_RAW）向程序设计人员提供了读写 IP/ICMP/IGMP 以及构造特殊的

IP 报文的功能。结合套接口选项 IP_HDRINCL 的使用，可以实现一些网络诊断、测试程序。使用原始套接口的基本步骤如下：

（1）以 AF_INET + SOCK_RAW + IPPROTO_XXX 为参数创建原始套接口，如下：

```
SOCKET sock = socket(AF_INET, SOCK_RAW, protocol);
```

其中 protocol 为 IPPROTO_XXX 形的协议名常量，如 IPPROTO_ICMP 等，一般不能设为 0。

（2）按照需要选择是否设置 IP_HDRINCL 选项。

（3）如果调用了 bind 函数，那么内核会将绑定的地址作为输出报文的源地址，绑定时的端口号对原始套接口没有任何意义；如果调用了 connect 函数，那么只有来自被 connect 的地址的报文才会被内核交付给原始套接口（对 connect 函数应用于无连接套接口的讨论见 17.6.6 节）。一般情况下，原始套接口不调用 bind 和 connect 函数。

（4）组装数据，如果未启用 IP_HDRINCL，那么应用进程只需构造 IP 头之外的数据；否则需要提供 IP 头，但是其中的 IPv4 标志字段与头部校验和可以留给内核填充。在 IP 头之外的数据，如果需要有校验和（例如 ICMP），那么必须由应用进程计算并提供。

（5）调用发送函数，如 sendto 发送数据。需要注意的是，在 IP_HDRINCL 选项启用时（如果未启用，那么内核会将 sendto 的目标地址自动填充在 IP 头的目的 IP 地址字段），如果 sendto 中的目标地址参数 to 与所填充的 IP 报头中的目的地址字段不同，那么应用程序不能正常工作。为何不能正常工作？用下一节中 ping 程序来做一个试验：在构造 IP 报头时填入 202.119.9.242 的目的地址，在调用 sendto 时 to 参数设为 202.119.9.99，截获的报文如图 19.6 所示。

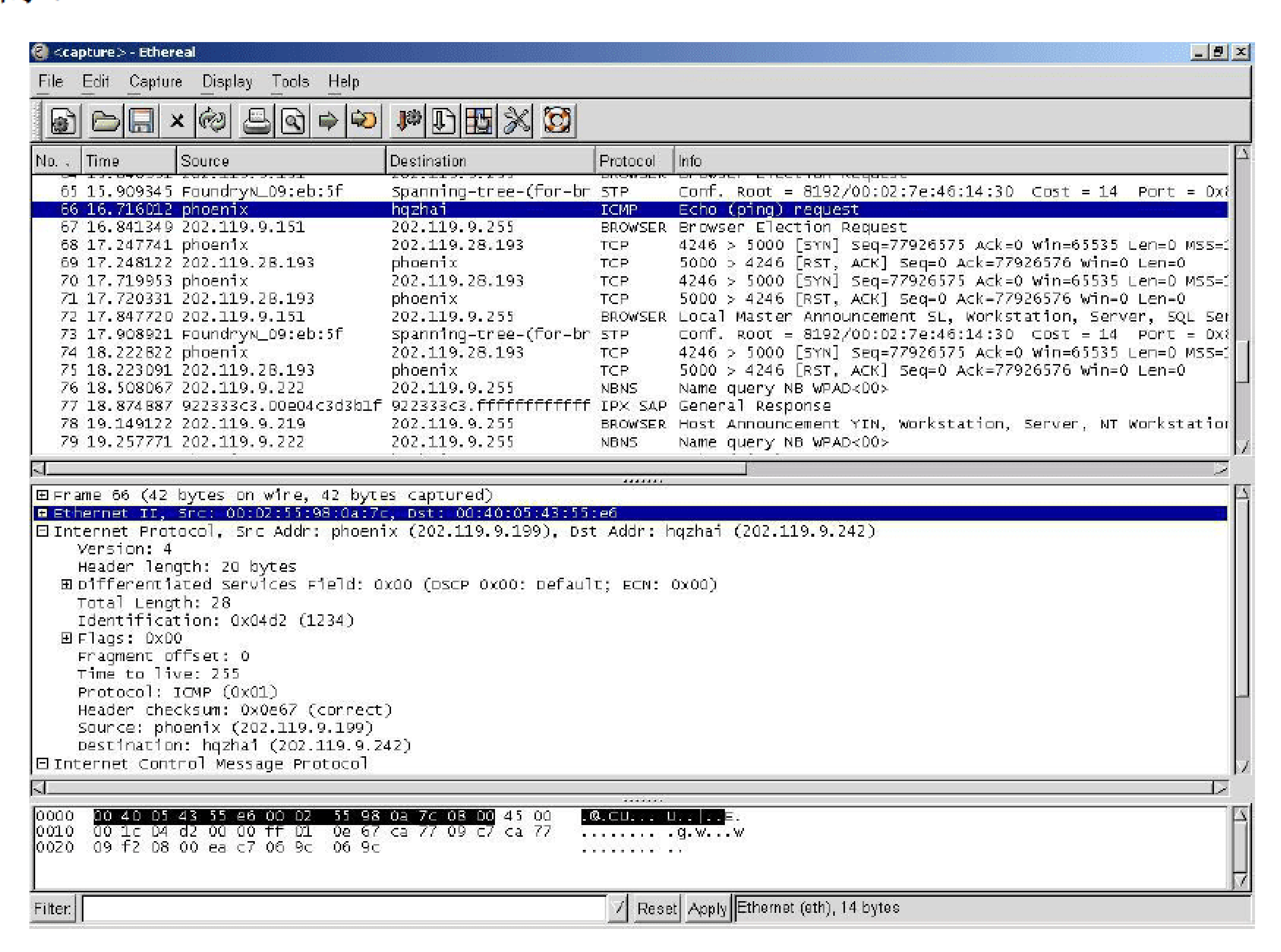


图 19.6　IP_HDRINCL 选项的测试

需要注意的是，在图 19.6 中 hqzhai 主机（即 202.119.9.242）的 MAC 地址为 00:40:05:43:55:e6。那么实际情况是怎么样的呢？首先使用 Windows 的 ping 命令：ping 202.119.9.242 / ping 202.119.9.99；然后输入命令 arp –a，查看到的正确的 IP-MAC 对应表如图 19.7 所示。

```
命令提示符
D:\Documents and Settings\Administrator>arp -a

Interface: 202.119.9.199 on Interface 0x1000003
  Internet Address      Physical Address      Type
  202.119.9.1           00-d0-d3-3a-67-dc     dynamic
  202.119.9.99          00-40-05-43-55-e6     dynamic
  202.119.9.242         00-06-29-5b-5a-21     dynamic

D:\Documents and Settings\Administrator>
```

图 19.7　真实的 MAC 地址

我们可以看出，在这种情况下应用进程产生的数据帧中的 MAC 地址是 to 地址对应的 MAC 地址，而目标 IP 地址是填充的 IP 头的数据，两者不匹配。这也就是此时程序不能正常工作的原因。

（6）调用接收函数如 recvfrom 接收数据。只有 ICMP、IGMP 以及内核不能识别协议字段的 IP 数据报会被传递给原始套接口。

下面两节以两个实用程序来介绍原始套接口的使用。

19.4.1　Ping 程序

ping 程序是典型的原始套接口的应用例子，通过它可以了解 IP_HDRINCL 选项的使用和如何组装 IP、ICMP 报头。程序的基本流程如下：

（1）以 AF_INET、SOCK_RAW 和 IPPROTO_ICMP 为参数创建原始套接口。

（2）启用 IP_HDRINCL 选项。

（3）构造自己的 IP 头及 ICMP 报文，需要注意的是 ICMP 报文的校验和必须由应用进程计算并提供。

（4）发送 ICMP Echo 报文，并解析、显示返回的 ICMP Echo Reply 报文。

```
****************************** 程序 19.3 Ping ******************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>
#include <WS2TCPIP.H>
#include <PROCESS.H>

// The IP header
typedef struct iphdr {
     unsigned char   ver_hlen;         // version & length of the header
```

```
    unsigned char   tos;            // Type of service, 8 位
    unsigned short  total_len;      // total length of the packet, 16 位
    unsigned short  ident;          // unique identifier, 16 位
    unsigned short  frag_and_flags; // flags, 16 位
    unsigned char   ttl;            // 8 位
    unsigned char   proto;          // protocol (TCP, UDP etc), 8 位
    unsigned short  checksum;       // IP checksum, 16 位
    unsigned int    sourceIP;       // 32 位, 源 IP 地址
    unsigned int    destIP;         // 32 位, 目的 IP 地址
}IPHeader;

// ICMP header structure
typedef struct icmphdr
{
    unsigned char      type;
    unsigned char      code;
    unsigned short     cksum;
    unsigned short     id;
    unsigned short     seq;
} ICMPHeader;

#define ICMP_ECHO_REPLY             0
#define ICMP_ECHO_REQUEST           8

#define PACKAGE_SIZE     sizeof(IPHeader) + sizeof(ICMPHeader)

#define xmalloc(s)       HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, (s))
#define xfree(p)         HeapFree (GetProcessHeap(), 0, (p))

void usage();
void HandleError(char *);
void FillPackage(char *, unsigned long, u_short);
USHORT CheckSum(USHORT *, int);

int main(int argc, char* argv[])
{
    unsigned long dstIP;
    BOOL bBroadcast = false;
    if(argc == 2){
        dstIP = inet_addr(argv[1]);
        if(dstIP == INADDR_NONE){
            usage();
```

```
                return -1;
            }
        }
        else
        if(argc == 3){
            if(strcmp(argv[1], "-b") != 0){
                usage();
                return -1;
            }
            bBroadcast = true;
            dstIP = inet_addr(argv[2]);
            if(dstIP == INADDR_NONE){
                usage();
                return -1;
            }
        }
        else{// argc != 2, 3
            usage();
            return -1;
        }

        int pid = _getpid();

        WSAData wsaData;
        WSAStartup(WINSOCK_VERSION, &wsaData);
        SOCKET sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
        if(sock == INVALID_SOCKET){
            HandleError("socket");
            WSACleanup();
            return -1;
        }

        BOOL on = TRUE;
        if(setsockopt(sock, IPPROTO_IP, IP_HDRINCL, (char *) &on, sizeof(on))
          == SOCKET_ERROR){
            HandleError("setsockopt");
            WSACleanup();
            return -1;
        }

        struct sockaddr_in to, from;
        memset(&to, 0, sizeof(to));
        to.sin_addr.s_addr = dstIP;
```

```
    to.sin_family = AF_INET;
    int len = sizeof(to);

    // 此处应尽量不调用 malloc，当用 malloc 分配内存，然后调用 socket API
    // 如 sendto 时会报 10004 错误，即该调用被系统中断（由 malloc 引起）
    char *buf = (char *) xmalloc(PACKAGE_SIZE);

    struct timeval tv;
    fd_set readSet;

    BOOL bError = false;
    for(int i = 0; i < 3 && !bError; i++){
        printf("ping %s ... %d\n", inet_ntoa(to.sin_addr), i+1);
        FillPackage(buf, dstIP, (u_short) pid);
        if(sendto(sock, buf, PACKAGE_SIZE, 0, (struct sockaddr *) &to,
          len) == SOCKET_ERROR){
            HandleError("sendto");
            break;
        }

        while(1){
            tv.tv_sec = 3;
            tv.tv_usec = 0;
            FD_ZERO(&readSet);
            FD_SET(sock, &readSet);

            int res = select(sock + 1, &readSet, NULL, NULL, &tv);
            if(res == SOCKET_ERROR){
                HandleError("select");
                bError = true;
                break;
            }
            if(res == 0){
                if(!bBroadcast)
                    printf("time out!\n");
                break;
            }

            if(FD_ISSET(sock, &readSet)){
                memset(buf, 0, PACKAGE_SIZE);
                memset(&from, 0, sizeof(from));
                len = sizeof(from);
                if(recvfrom(sock, buf, PACKAGE_SIZE,0, (struct
```

```
                    sockaddr *) &from, &len) == SOCKET_ERROR){
                        HandleError("recvfrom");
                        bError = true;
                        break;
                    }

                    IPHeader *pIPHdr = (IPHeader *) buf;
                    ICMPHeader *pICMPHdr = (ICMPHeader *) (buf +
                      sizeof(IPHeader));
                    if(pICMPHdr->id == htons((u_short) pid)
                        && pICMPHdr->seq == htons((u_short) pid)
                        && pICMPHdr->type == ICMP_ECHO_REPLY){
                        printf("Echo Reply From %s.\n",
                          inet_ntoa(from.sin_addr));
                        if(!bBroadcast) break;
                    }
                }
            }// end of while
        }// end of for

        xfree(buf);

        closesocket(sock);
        WSACleanup();

        return 0;
}

void FillPackage(char *pData, unsigned long dstIP, u_short id)
{
        memset(pData, 0, PACKAGE_SIZE);

        IPHeader* pIPHeader=(IPHeader*) pData;

        int nVersion = 4;
        int nHeadSize = sizeof(IPHeader) / 4;
        unsigned long srcIp = inet_addr("202.119.9.199");//此处不能使用
          INADDR_ANY;
        unsigned long destIp = dstIP;
        pIPHeader->ver_hlen = (nVersion << 4) | nHeadSize;
        pIPHeader->tos = 0;
        pIPHeader->total_len = htons(PACKAGE_SIZE);
        pIPHeader->ident = htons(1234);
```

```
    pIPHeader->frag_and_flags = 0;
    pIPHeader->ttl = 255;
    pIPHeader->proto = IPPROTO_ICMP;
    pIPHeader->checksum = 0;
    pIPHeader->sourceIP = srcIp;
    pIPHeader->destIP = destIp;
    /* the below can be ignored */
    pIPHeader->checksum = CheckSum((USHORT*) pData, sizeof(IPHeader));

    ICMPHeader* pICMPHeader=(ICMPHeader*) (pData + sizeof(IPHeader));
    pICMPHeader->type = ICMP_ECHO_REQUEST;
    pICMPHeader->code = 0;
    pICMPHeader->cksum = 0;
    pICMPHeader->id = htons(id);
    pICMPHeader->seq = htons(id);
    pICMPHeader->cksum = CheckSum((USHORT*)((char*)pData + sizeof
      (IPHeader)), sizeof(ICMPHeader));
}

USHORT CheckSum(USHORT *pUShort, int size)
{
    unsigned long cksum=0;

    while(size > 1)
    {
        cksum += *pUShort++;
        size -= sizeof(USHORT);
    }
    if(size){// size == 1
        cksum += *(UCHAR*)pUShort;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >>16);

    return (USHORT)(~cksum);
}

void usage()
{
    printf("usage: ping [-b] hostIP\n");
    printf("notes: 如果ping广播地址必须加-b,否则无法得到完整输出.\n");
}
```

```
192 void HandleError(char *func)
193 {
194         int errCode = WSAGetLastError();

195         char info[65] = {0};
196         _snprintf(info, 64, "%s:          %d\n", func, errCode);
197         printf(info);
198 }
```

```
***************************************************************************
```

第 6～18 行定义了 IP 报文头结构 IPHeader。

第 19～27 行定义了 ICMP 报文头结构 ICMPHeader。

第 28～29 行定义了两个常量 ICMP_ECHO_REPLY 和 ICMP_ECHO_REQUEST，分别对应 ping 的应答报文和 ping 的请求报文。

第 39～64 行读取 ping 程序的输入参数，包含目标地址 dstIP 和“-b”广播标志 bBroadcast。

第 65 行获取当前进程的 ID 号，并将其赋值给 pid。

第 66～67 行初始化 winsock。

第 68～73 行创建 ICMP 协议类型的原始套接口。

第 74～79 行调用 setsockopt 函数，启用 IP_HDRINCL 选项。

第 93～136 行连续发送三次 ping 请求报文，并输出其接收到的应答。

- ❐ 95～99 行，调用 FillPackage 填充 ICMP 报文，然后将其发送。
- ❐ 100～135 行，while(1)循环，用于接收返回的应答报文。之所以需要在循环体内接收报文，是因为本程序在设置-b 输入参数后就可以向全网段发送 ping 请求报文，一次请求能得到多个应答报文。
  - ✧ 101～115 行，应用 select 模型进行读操作。当 select 超时说明不再有响应报文时，退出 while 循环。
  - ✧ 117～134 行，解析数据，并输出确认为 ICMP 应答的报文。

第 138～141 行释放资源，结束 winsock 的使用并退出程序。

第 143～171 行 FillPackage 函数，根据输入的目标 IP 和进程号参数，组装 ping 请求报文。

第 172～186 行 CheckSum 函数，计算报文的校验值。

19.4.2 WinSniffer 程序

WinSniffer 是一个简单的网络监视器原型，用于演示原始套接口和 WSAIoctl 函数中 SIO_RCVALL 选项的使用。运行该程序后，将以“协议名 From 源地址”的格式输出本机指定网络接口上接收到的所有报文。

WSAIoctl 函数是除 ioctlsocket、setsockopt/getsockopt 之外的另一个套接口模式/选项配置函数，用于设置或者读取套接口上的操作选项参数。函数定义如下：

```
int WSAIoctl(
```

```
    SOCKET s,
    DWORD dwIoControlCode,
    LPVOID lpvInBuffer,
    DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

其中 s 表示待处理的套接口对象；dwIoControlCode 为控制码；lpvInBuffer 是输入缓冲区指针，可能是布尔型、整型，也可能是结构体与 dwIoControlCode 对应；cbInBuffer 是 lpvInBuffer 的大小字节数；lpvOutBuffer 是输出缓冲区；cbOutBuffer 是 lpvOutBuffer 的大小；lpcbBytesReturned 用于保存实际的输出字节数；lpOverlapped 和 lpCompletionRoutine 仅用于重叠套接口，分别指向重叠结构和完成例程。

从 Windows 2000 起，Winsock2 新增了很多 WSAIoctl 操作码如 SIO_RCVALL、SIO_RCVALL_MCAST、SIO_RCVALL_IGMPMCAST、SIO_KEEPALIVE_VALS、SIO_ABSORB_RTRALERT、SIO_UCAST_IF、SIO_LIMIT_BROADCASTS、SIO_INDEX_BIND、SIO_INDEX_MCASTIF、SIO_INDEX_ADD_MCAST、SIO_INDEX_DEL_MCAST。WinSniffer 应用了其中的 SIO_RCVALL 选项，使套接口可接收流经本地指定网络接口的所有 IP 报文。该选项只适用于 AF_INET 地址族+原始套接口+IPPROTO_IP 协议，并且套接口必须明确指定需绑定的本地网络接口，也就是说不能使用通配网址 INADDR_ANY。

程序的基本流程是：

（1）以 AF_INET、SOCK_RAW 和 IPPROTO_IP 为参数创建原始套接口。

（2）绑定本地地址 202.119.9.199。

（3）调用 WSAIoctl 设置 SIO_RCVALL 为 TRUE。

（4）接收、解析报文并输出分析结果；需要注意的是，由于我们只关心 IP 报头，程序只分配了很小的缓冲区，因此在接收报文时 recvfrom 函数返回 WSAEMSGSIZE 错误是完全正常的。

由于程序相对较为简单，就不再对它进行详细分析。

```
**************************程序19.4 Winsniffer****************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>
#include <MSTcpIP.h>

// The IP header
typedef struct iphdr {
    unsigned char   ver_hlen;        // version & length of the header
    unsigned char   tos;             // Type of service, 8位
    unsigned short  total_len;       // total length of the packet, 16位
```

```
    unsigned short  ident;          // unique identifier, 16位
    unsigned short  frag_and_flags; // flags, 16位
    unsigned char   ttl;            // 8位
    unsigned char   proto;          // protocol (TCP, UDP etc), 8位
    unsigned short  checksum;       // IP checksum, 16位
    unsigned intsourceIP;           // 32位, 源IP地址
    unsigned intdestIP;             // 32位, 目的IP地址
}IPHeader;

#define PACKAGE_SIZE   sizeof(IPHeader)
#define xmalloc(s)     HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, (s))
#define xfree(p)       HeapFree (GetProcessHeap(), 0, (p))

BOOL WINAPI CtrlHandler(DWORD dwEvent);
void HandleError(char *);

BOOL g_bExit = FALSE;

int main(int argc, char* argv[])
{
    if(!SetConsoleCtrlHandler(CtrlHandler, TRUE)){
        printf("SetConsoleCtrlHandler: %d\n", GetLastError());
        return -1;
    }

    WSAData wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);
    SOCKET sock = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
    if(sock == INVALID_SOCKET){
        HandleError("socket");
        WSACleanup();
        return -1;
    }

    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_addr.s_addr = inet_addr("202.119.9.199");
    addr.sin_family = AF_INET;
    if(bind(sock, (struct sockaddr *) &addr, sizeof(addr)) == SOCKET_ERROR){
        HandleError("bind");
    }

    int on = RCVALL_ON;
    DWORD num;
    if(WSAIoctl(sock, SIO_RCVALL, &on, sizeof(on), NULL, 0, &num, NULL, NULL)
    == SOCKET_ERROR){
        HandleError("WSAIoctl Set");
    }
```

```
    char *buf = (char *) xmalloc(PACKAGE_SIZE);
    struct sockaddr_in from;
    int fromlen;

    while(!g_bExit){
        memset(buf, 0, PACKAGE_SIZE);
        memset(&from, 0, sizeof(from));
        fromlen = sizeof(from);
    if(recvfrom(sock, buf, PACKAGE_SIZE, 0, (struct sockaddr *) &from, &fromlen)
    == SOCKET_ERROR){
            if(WSAGetLastError() != WSAEMSGSIZE){
                HandleError("recvfrom");
                break;
            }
        }

        switch(((IPHeader *) buf)->proto){
        case IPPROTO_ICMP:
            printf("ICMP From %s\n", inet_ntoa(from.sin_addr));
            break;
        case IPPROTO_IGMP:
            printf("IGMP From %s\n", inet_ntoa(from.sin_addr));
            break;
        case IPPROTO_TCP:
            printf("TCP From %s\n", inet_ntoa(from.sin_addr));
            break;
        case IPPROTO_UDP:
            printf("UDP From %s\n", inet_ntoa(from.sin_addr));
            break;
        default:
            printf("UnKnown datagram From %s\n", inet_ntoa(from.sin_addr));
        }

    }

    xfree(buf);

    closesocket(sock);
    WSACleanup();
    printf("Stopped!\n");

    return 0;
}

BOOL WINAPI CtrlHandler(DWORD dwEvent)
{
    switch(dwEvent){
```

```
    case CTRL_C_EVENT:
    case CTRL_LOGOFF_EVENT:
    case CTRL_SHUTDOWN_EVENT:
    case CTRL_CLOSE_EVENT:
        printf("Stopping......\n");
        g_bExit = TRUE;
        break;
    default:
     return FALSE;
}

    return TRUE;
}

void HandleError(char *func)
{
    int errCode = WSAGetLastError();

    char info[65] = {0};
    _snprintf(info, 64, "%s:        %d\n", func, errCode);
    printf(info);
}
```

****

第 20 章　UDP 服务器编程

相对于服务器端编程而言，客户端的程序设计比较简单，因此本书以服务器编程介绍为主。从第 20 章起，我们讨论几种常见的 UDP/TCP 服务器的系统架构设计，并对其进行性能分析。在介绍具体的服务器设计之前，首先在 20.1 节简单地介绍多线程编程的基本知识，在此基础上分别介绍迭代 UDP 服务器、并发 UDP 服务器、迭代 TCP 服务器和并发服务器的设计方法。

20.1　多线程编程

需要指出的是，本书的主要目标是介绍 Winsock 网络编程知识，多线程编程只是网络编程的辅助技术，因此我们并不打算加以详细的介绍。20.1 节仅仅告诉读者如何创建一个线程，以及如何进行简单的线程同步。

20.1.1　线程的创建

Windows API 提供了多个函数接口以创建线程，本书首先介绍其中的_beginthread 函数。

```
unsigned long _beginthread( void( __cdecl *start_address )( void * ),
                            unsigned stack_size,
                            void *arglist );
```

- start_address：输入参数，待创建的新线程所要执行的工作函数的起始地址。
- stack_size：输入参数，新线程的栈的大小，也可以设置为 0。
- arglist：需要传递给新线程的参数列表，可以为 NULL。
- 返回值：如果线程创建成功，函数返回新创建线程的句柄；否则返回–1，此时系统会自动设置 errno 值。

使用_beginthread 函数创建线程是非常简便的。首先定义一个工作线程函数，原型如下：

```
void WorkThread(LPVOID lpParam);
```

该函数的返回值必须是 void。所有的线程工作函数都有一个 32 位的输入参数，应用程序可以用它来传递数值，如 accept 返回的 SOCKET 变量（本质上是 u_int 类型），但更多地是传递一个指向结构体的指针。随后，我们就可以创建线程了：

```
HANDLE hThread = (HANDLE) _beginthread(WorkThread, 0, hIOCP);
```

如果创建线程成功，函数会返回新线程的句柄。

我们可以调用_endthread 函数来结束线程，但是一般不需要这么做：当线程函数执行完



毕时会自动调用该函数。_endthread 函数会释放线程占用的资源并自动关闭线程句柄，因此必须牢记：对于_beginthread 函数创建的线程，不要调用 CloseHandle 函数来关闭返回的句柄。这一点与第 21 章介绍到的 CreateThread 函数有所不同。

20.1.2 线程的同步

Windows 向程序设计人员提供了多种线程同步方法，如 mutex、semaphore、事件对象以及临界区对象等，本节介绍其中的两种。

（1）事件对象（Event Objects）机制

Win32 应用程序可以使用事件对象机制来通知等待线程某个条件已满足。以一个简单的例子来说明这种机制的应用：假设有三个线程——主线程、ReadThread 和 WriteThread，同步要求是 ReadThread 必须在 WriteThread 的写操作完成之后才能进行读操作，主线程必须在 ReadThread 的读操作完成后才结束。

定义两个事件对象 evToRead 和 evToFin，前者由 WriteThread 用于通知 ReadThread 进行读操作，后者由 ReadThread 用于通知主线程读操作结束。事件对象的创建、设置及等待可参见 18.2.3 节介绍。图 20.1 是线程同步的示意图。

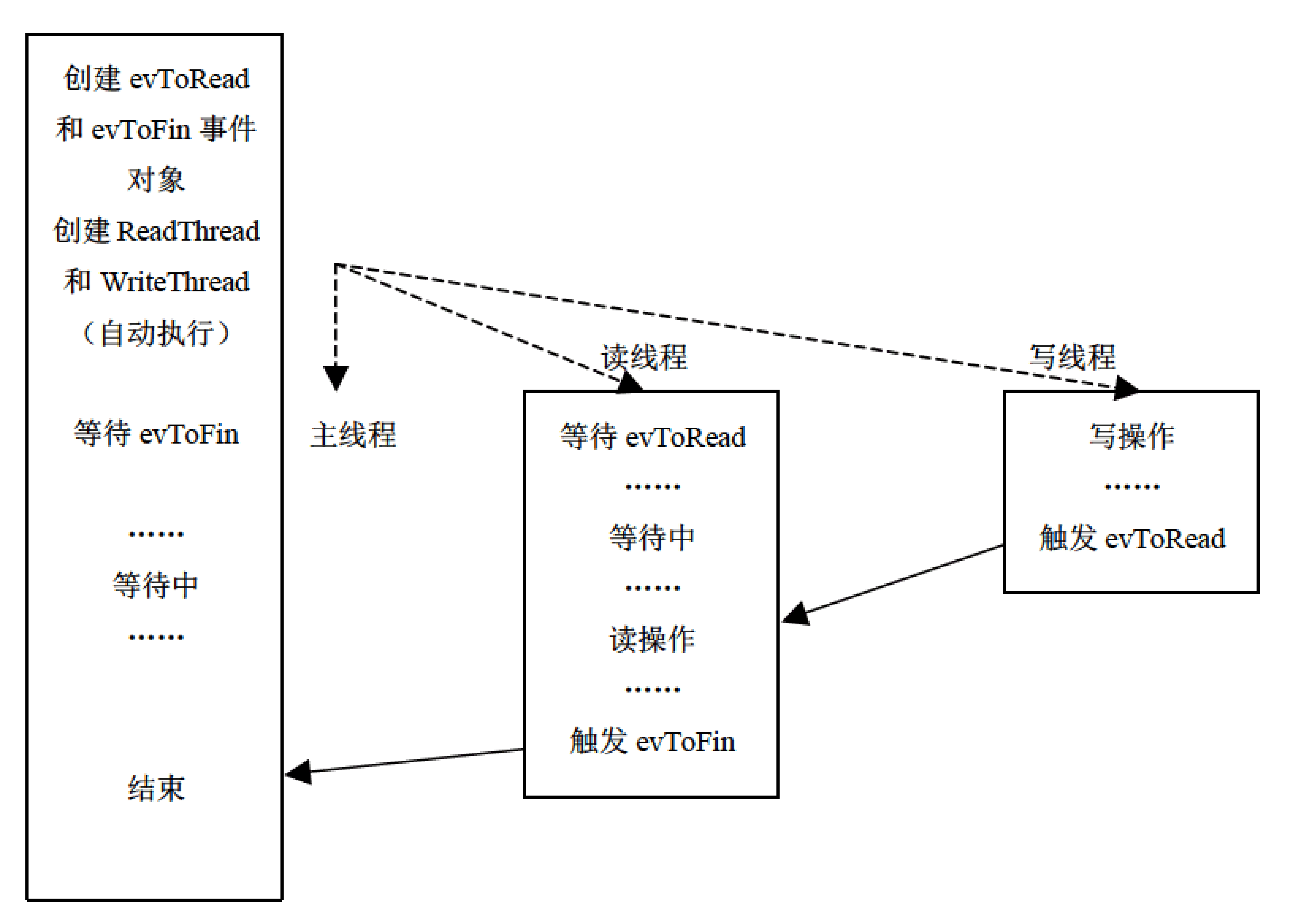


图 20.1　线程同步要求

程序实现如下：

```
*************************** 程序20.1 SyncEvent **************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <PROCESS.H>
#include <WINSOCK2.H>
```

```
WSAEVENT evToRead, evToFin;

void ReadThread(LPVOID param)
{
    WSAWaitForMultipleEvents(1, &evToRead, TRUE, WSA_INFINITE, FALSE);
    printf("Reading...\n");
    for(int i = 9; i >= 0; i--){
        printf("%x ", i);
        Sleep(500);
    }
    printf("\nRead Finished!\n");
    WSASetEvent(evToFin);
}
void WriteThread(LPVOID param)
{
    printf("Writing...\n");
    for(int i =0; i < 10; i++){
        printf("%x ", i);
        Sleep(500);
    }
    printf("\nWrite Finished!\n");
    WSASetEvent(evToRead);
}

int main(int argc, char* argv[])
{
    printf("Main Start ->\n");
    evToRead = WSACreateEvent();
    evToFin = WSACreateEvent();
    _beginthread(ReadThread, 0, NULL);
    _beginthread(WriteThread, 0, NULL);
    WSAWaitForMultipleEvents(1, &evToFin, TRUE, WSA_INFINITE, FALSE);
    printf("<-Main Finnish\n");

    return 0;
}
************************************************************************
```

程序的输出如图 20.2 所示。

```
"E:\My Documents\文档\《TCPIP与网络编程》\Code\SyncEvent\Release\SyncEvent.exe"
Main Start ->
Writing...
0 1 2 3 4 5 6 7 8 9
Write Finished!
Reading...
9 8 7 6 5 4 3 2 1 0
Read Fininshed!
<-Main Finnish
Press any key to continue
```

图 20.2　SyncEvent 输出

（2）临界区对象（Critical Section Objects）机制

另一种简单并且常用的线程同步机制是临界区对象。与事件对象相比，临界区更适用于多个线程操作之间没有先后顺序但要求互斥的同步。

要使用临界区对象机制，首先必须定义 CRITICAL_SECTION 变量，该变量是所有临界区操作的对象。

```
CRITICAL_SECTION CS;
```

定义了变量 CS 之后，在实际使用之前还需要调用 InitializeCriticalSection 对 CS 进行初始化。该函数只有一个输出类型的参数 LPCRITICAL_SECTION，没有返回值，但是在内存不足的情况下会抛出 STATUS_NO_MEMORY 异常。

```
InitializeCriticalSection(&CS);
```

初始化了 CS 之后，EnterCriticalSection、TryEnterCriticalSection 以及 LeaveCriticalSection 函数就可以为多个线程提供共享资源的互斥访问了。当线程调用 EnterCriticalSection 函数时，系统会判断 CS 对象是否已被锁定，如果没有被锁定，那么线程就可以进入临界区以进行共享资源的互斥访问并且同时 CS 被置为锁定状态；否则，说明有线程已在使用共享资源，调用线程将被阻塞以等待 CS 解锁。如果线程不希望被阻塞而只是想尝试进入，那么可以调用 TryEnterCriticalSection 函数。在对共享资源的互斥访问结束后，线程应该调用 LeaveCriticalSection 函数来解锁 CS，此时其他线程中就能进入临界区了。

```
EnterCriticalSection(&CS);
// Access the shared resource.
……
// Release ownership of the critical section.
LeaveCriticalSection(&CS);
```

在进程中的各线程结束同步操作后，进程必须调用 DeleteCriticalSection 函数以释放系统资源。

```
DeleteCriticalSection(&CriticalSection);
```

把涉及到的这些函数的定义列表如下：

```
VOID InitializeCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection  // critical section
);
BOOL TryEnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection  // critical section
);
VOID EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection  // critical section
);
VOID LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection   // critical section
);
VOID DeleteCriticalSection(
```

```
  LPCRITICAL_SECTION lpCriticalSection   // critical section
);
```

20.2　迭代服务器

迭代服务器的设计相对于并发服务器来说比较简单，对于 UDP 来说更为如此。多数 UDP 服务器程序都是迭代执行：服务器等待请求，接受请求，处理请求，送回应答，然后再次等待请求……典型的例子如 ECHO 服务。如果处理请求处理过程较为复杂，需要服务器－客户端之间的多次交互，那么在此交互过程中，所有来自不同源地址的报文都将被丢弃。在这种架构下，服务器应注意在等待客户端的非起始请求报文时，应进行严格的超时控制，以防止服务器长期地停留在为某个客户端服务的等待中。

给出一个服务器与客户端之间需要两步数据交互的例子，如图 20.3 所示。

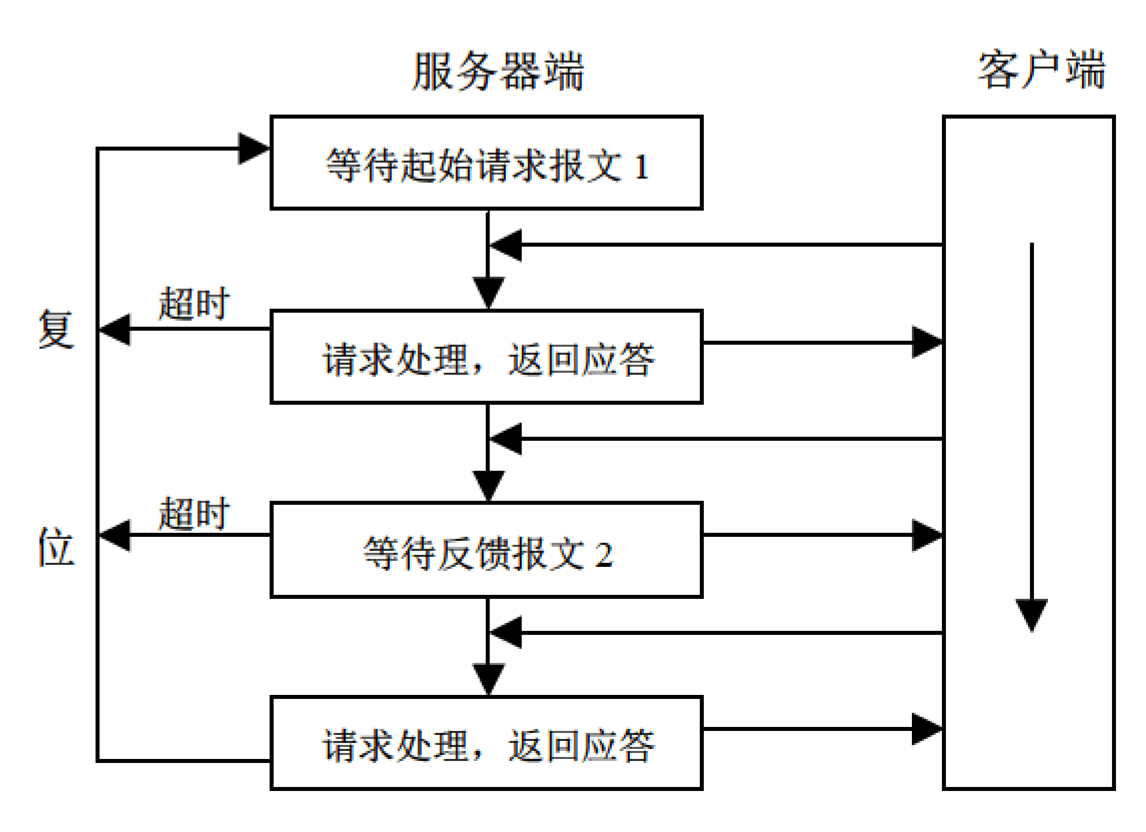


图 20.3　客户－服务器的交互

服务器运行过程如下：① 服务器等待起始请求报文 1；② 当接收到报文 1 后，将客户端的源地址－端口号记录在案，处理请求并传回应答；③ 服务器等待客户端的反馈报文 2，如果超时，服务器复位；如果收到的报文并非来自起始报文的源地址－端口，那么丢弃该报文（也可使用 connect 函数来完成这个功能）；否则进入步骤④处理请求，返回应答；⑤ 复位，服务器回至起始状态。

20.3　并发服务器

在迭代服务器的介绍中提到，如果服务器－客户端之间的交互较为复杂，那么来自其他客户端的请求就会被丢弃，这在某些应用情况下不可接受到，因此就需要用到并发服务器设计。

典型的解决方法是这样的：服务器等待起始的请求报文；当有客户服务请求到来时，创建一个新的套接口；将该套接口的端口号作为服务器的第一个应答，要求客户端随后与该套

接口交互。借鉴 TCP 服务器，把服务器的接受请求的套接口称为监听套接口，而把创建的新套接口称为服务套接口，其流程如图 20.4 所示。

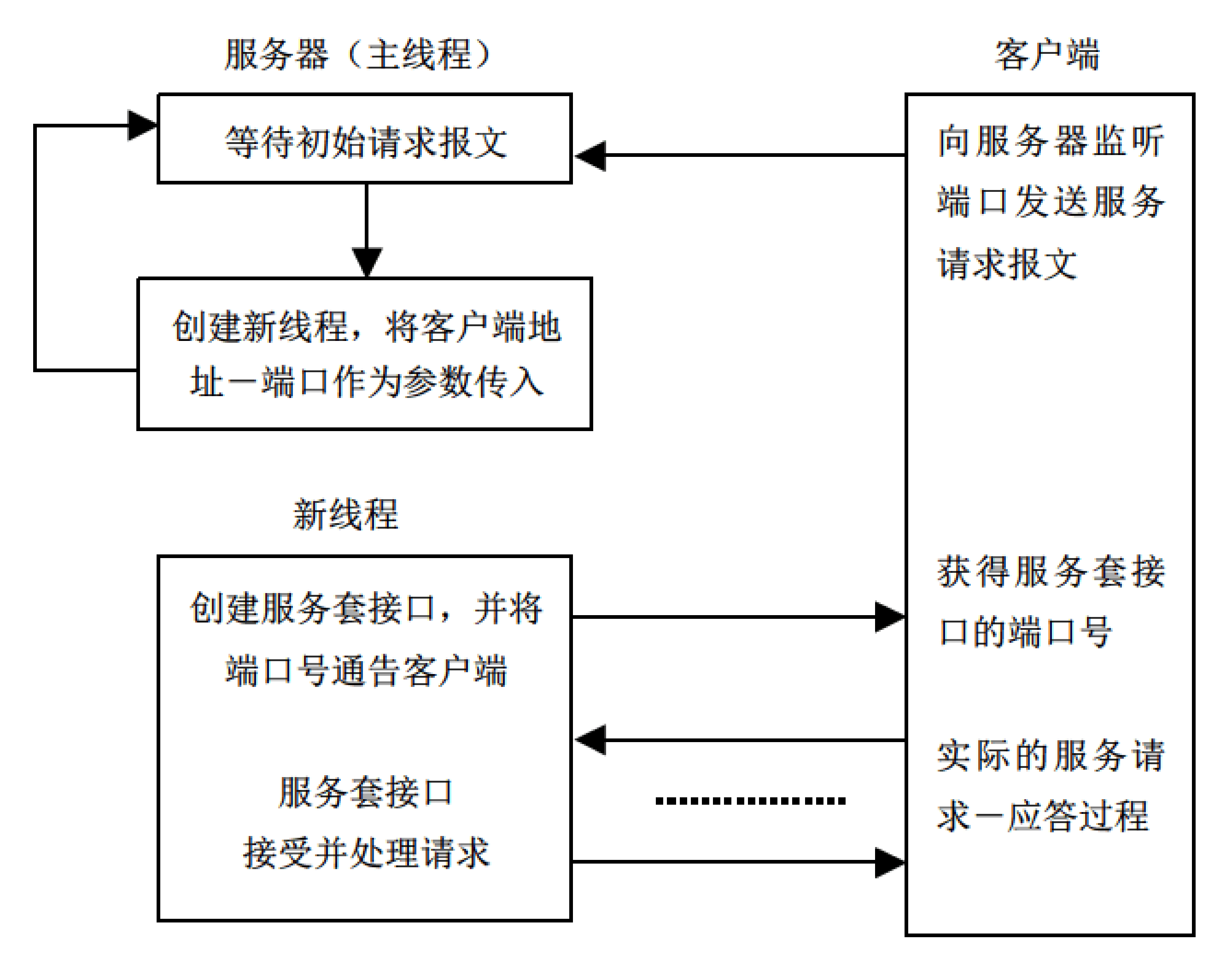


图 20.4　并发 UDP 服务器

一个简单的服务器程序如下：

```
**************************** 程序 20.2 UDPCoSvr **************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <TIME.H>
#include <WINSOCK2.H>
#include <PROCESS.H>

typedef struct _UserDATA{
     byte type; //  报文类型, 0:初始请求 1:应答-ServerSocketAddress 2:请求
       3:应答-echo
     char buf[500];
}UserData;

void HandleError(char *func)
{
#ifdef _DEBUG
     int errCode = WSAGetLastError();

     char info[65] = {0};
     _snprintf(info, 64, "%s:          %d\n", func, errCode);
```

```
    printf(info);
#endif
}

void ServerThread(LPVOID param)
{
    time_t tm;
    time(&tm);

    printf("Server Thread %d started.\n", tm);

    UserData data;
    struct sockaddr_in *fromaddr = (struct sockaddr_in *) param;
    SOCKET sockSvr = socket(AF_INET, SOCK_DGRAM, 0);
    if(sockSvr == INVALID_SOCKET){
        HandleError("socket");
        return;
    }

    if(connect(sockSvr, (struct sockaddr *) fromaddr, sizeof(struct
      sockaddr_in)) == SOCKET_ERROR){
        HandleError("connect");
        closesocket(sockSvr);
        return;
    }
    HeapFree(GetProcessHeap(), 0, fromaddr);

    // 应答 1
    memset(&data, 0, sizeof(data));
    data.type = 1;
    // client can get the server socket address when recv the datagram
    if(send(sockSvr, (char *) &data, sizeof(data), 0) == SOCKET_ERROR){
        HandleError("send");
        closesocket(sockSvr);
        return;
    }
    // 请求 2
    timeval tv;
    memset(&tv, 0, sizeof(tv));
    tv.tv_sec = 5;
    fd_set readSet;
    FD_ZERO(&readSet);
    FD_SET(sockSvr, &readSet);
```

```
    int ret = select(0, &readSet, NULL, NULL, &tv);
    // CASE 1: select Error
    if(ret == SOCKET_ERROR){
        HandleError("select");
        closesocket(sockSvr);
        return;
    }
    // CASE 2: select Timeout
    if(ret == 0){
        closesocket(sockSvr);
        printf("Time over, server socket closed & thread %d finished!\n", tm);
        return;
    }
    // CASE 3: select OK
    if(FD_ISSET(sockSvr, &readSet)){
        memset(&data, 0, sizeof(data));
        ret = recv(sockSvr, (char *) &data, sizeof(data), 0);
        if(ret == SOCKET_ERROR){
            HandleError("recv");
            closesocket(sockSvr);
            return;
        }
    }
    else{
        closesocket(sockSvr);
        return;
    }

    if(data.type != 2){
        printf("Wrong Packet, Server socket closed & thread %d
          finished!\n", tm);
        closesocket(sockSvr);
        return;
    }
    // 应答 3 - echo
    data.type = 3;
    send(sockSvr, (char *) &data, sizeof(data), 0);

    closesocket(sockSvr);

    printf("Server Thread %d finished.\n", tm);
}
```

```
int main(int argc, char* argv[])
{
    WSAData wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);
    SOCKET sockListen = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in listenaddr;
    memset(&listenaddr, 0, sizeof(listenaddr));
    listenaddr.sin_family = AF_INET;
    listenaddr.sin_port = htons(9999);
    listenaddr.sin_addr.s_addr = INADDR_ANY;
    if(bind(sockListen, (struct sockaddr *) &listenaddr,
       sizeof(listenaddr)) == SOCKET_ERROR){
        HandleError("bind");
        return -1;
    }

    UserData initReq;
    struct sockaddr_in *from;
    int fromlen, ret;
    while(1){
        from = (struct sockaddr_in *) HeapAlloc(GetProcessHeap(),
                 HEAP_ZERO_MEMORY, sizeof(struct sockaddr_in));
        memset(&initReq, 0, sizeof(initReq));
        memset(from, 0, sizeof(struct sockaddr_in));
        fromlen = sizeof(struct sockaddr_in);
        ret = recvfrom(sockListen, (char *) &initReq, sizeof(initReq),
             0, (struct sockaddr *) from, &fromlen);
        if(ret == SOCKET_ERROR){
            HandleError("recvfrom");
            break;
        }
        if(ret != sizeof(initReq)){
#ifdef _DEBUG
            printf("Wrong package! Discarded!\n");
#endif
            continue;// discard
        }
        if(initReq.type != 0){
#ifdef _DEBUG
            printf("Wrong package type! Discarded!\n");
#endif
            continue;// discard
        }
```

```
131             _beginthread(ServerThread, 0, from);
132         }

133         return 0;
134 }
```

****

第 6～9 行 UserData 结构定义，其中 type 域用于指示报文类型（0 表示客户端的初始请求，1 表示服务器反馈的服务套接口地址，2 表示客户端请求 3 表示服务器的应答 echo）；buf 域作为数据区。

第 92～134 行 main 函数。

- 94～96 行，初始化 winsock，并创建 UDP 服务器套接口 sockListen，该套接口将用于接收所有的客户端初始请求报文。
- 97～105 行，将服务套接口绑定本地的 9999 端口（该端口必须是客户端知道的）。
- 109～132 行，循环接收来自各个客户端的初始请求报文，并将客户端的地址作为参数创建服务线程 ServerThread。

第 19～91 行 ServerThread 线程函数。

- 25 行，根据线程参数 param 获取该线程所有服务的客户端的地址 fromaddr。
- 26～35 行，为该客户端创建服务套接口 sockSvr，并调用 connect 函数在 sockSvr 和客户端地址之间建立绑定关系。这样做有两个优点：一是避免 sockSvr 接收到非该客户端地址的 UDP 报文；二是可以直接调用 send 和 recv 函数，而不是 sendto 和 recvfrom。
- 36 行，释放 fromaddr 所占资源。需要注意的是，该内存资源在主线程中分配，在 ServerThread 子线程中被释放。
- 37～45 行，发送应答报文（类型 1）。由于客户端在调用 recvfrom 函数时可以获得服务套接口的地址，因此不需要显示地将地址信息填充在应答报文中。
- 46～85 行，应用 select 函数接收客户端的后续请求报文（类型 2），并进行相应的报文格式判断。
- 86～88 行，服务套接口返回应答报文（类型 3）。
- 89～90 行，关闭服务套接口，并结束服务线程。

相对应的客户端程序片断如下，需要注意的是这些代码缺少必要的错误处理以及安全考虑，例如接收到的临时服务套接口的 IPv4 地址应该与监听套接口地址一致等。

```
SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in listenaddr, svraddr;// 分别对应服务器监听套接口和临时服务套接
    口地址
int len;
memset(&listenaddr, 0, sizeof(listenaddr));// 监听套接口地址
listenaddr.sin_family = AF_INET;
listenaddr.sin_port = htons(9999);
listenaddr.sin_addr.s_addr = inet_addr("202.119.9.199");
```

```
UserData data;
memset(&data, 0, sizeof(data));
data.type = 0;// 服务请求起始报文
sendto(sock, (char *) &data, sizeof(data), 0, (struct sockaddr *) &listenaddr,
    sizeof(listenaddr));
memset(&data, 0, sizeof(data));
len = sizeof(svraddr);
recvfrom(sock, (char *) &data, sizeof(data), 0, (struct sockaddr *) &svraddr,
    &len);
if(data.type == 1){// 获取了临时服务套接口地址
    printf("%s: %d\n", inet_ntoa(svraddr.sin_addr), ntohs(svraddr.sin_port));
}
connect(sock, (struct sockaddr *) &svraddr, sizeof(svraddr));
memset(&data, 0, sizeof(data));
data.type = 2;// 服务请求报文
strncpy(data.buf, "Hello! It's only a test!", 500);
send(sock, (char *) &data, sizeof(data), 0);
memset(&data, 0, sizeof(data));
recv(sock, (char *) &data, sizeof(data), 0);// 接收服务响应-Echo
if(data.type == 3){
    printf("Server Echo: %s\n", data.buf);
}
```

第 21 章　TCP 服务器编程

与 UDP 服务器编程相比，TCP 服务器编程要复杂得多，可以选择的系统结构设计也多种多样。本章首先简单地介绍迭代 TCP 服务器，然后介绍 3 种常用的并发 TCP 服务器系统设计。

21.1　迭代服务器

迭代 TCP 服务器系统结构非常简单，它的基本流程如下：

（1）创建监听流套接口。

（2）绑定本地的服务器知名端口。

（3）调用 listen 函数，使套接口处于监听状态。

（4）接受外来连接请求，accept 函数返回服务套接口。

（5）与客户端进行数据交互。

（6）关闭服务套接口，返回步骤（4）。关闭操作可能由以下的任意一种情况引发：

- 数据交互完毕，服务器主动关闭连接。
- 客户端已关闭连接。
- 服务器端接收数据超时。

如图 21.1 所示为迭代 TCP 服务器系统流程。

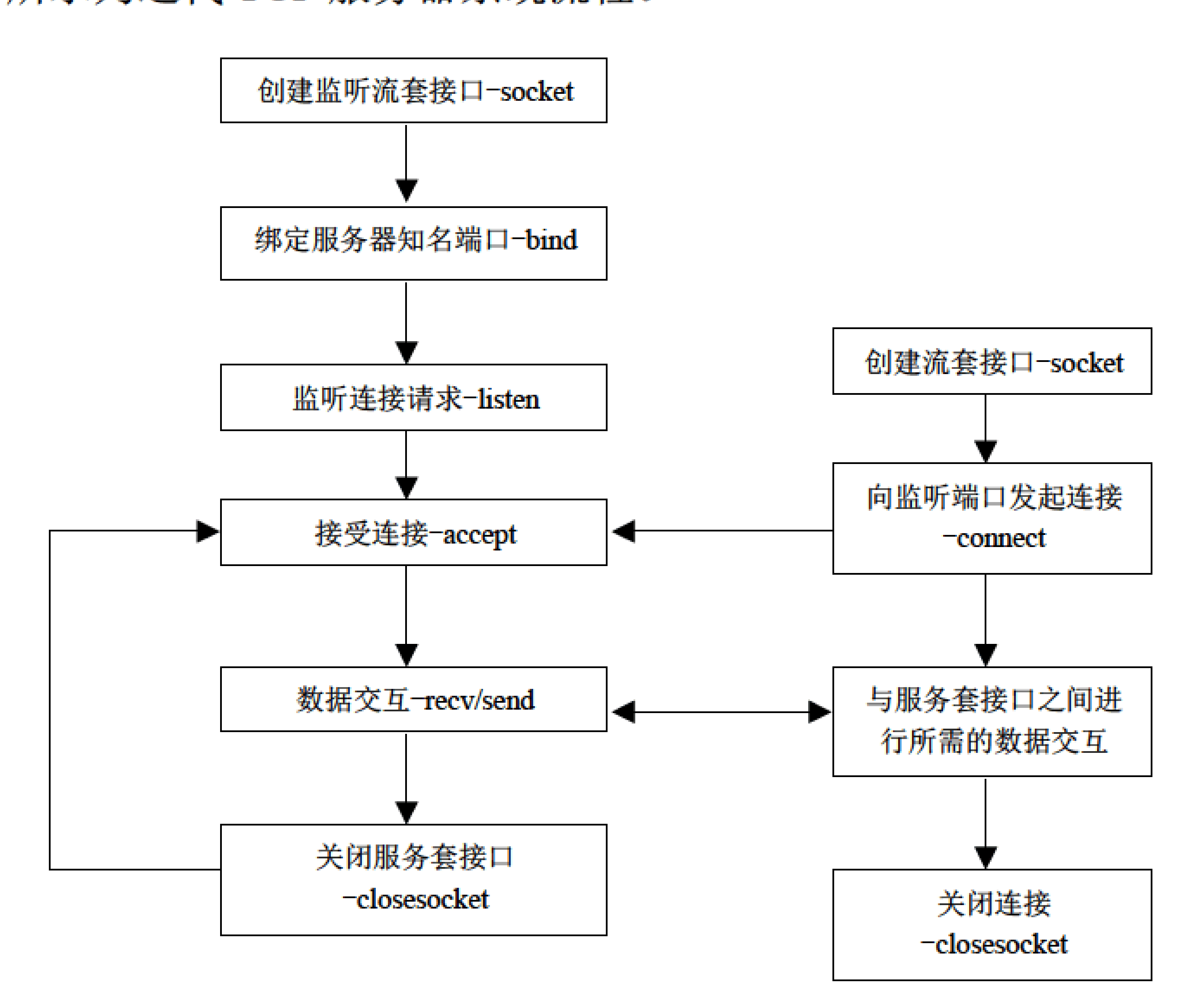


图 21.1　迭代 TCP 服务器系统流程

迭代服务器的最大问题是它无法处理多个、同时发生的客户端请求。如果有两个客户同时发送连接请求，那么其中一个必须等待直到另一个请求处理完毕。因此，迭代服务器架构通常只能用于一些简单的服务类型，如 Echo、Ping 等。

21.2　并发服务器

由于迭代服务器的种种局限性，人们通常采用另一种服务器架构——并发服务器。它与迭代服务器的最大区别是可以同时处理多个客户端的请求。

21.2.1　每客户单线程

在一般的并发服务器架构中，一个单独的线程等待客户端的连接请求，当有请求到来时该线程创建一个新的服务线程来进行处理，在新线程创建完毕后主线程再回复至等待请求状态。这种服务器架构也就是通常说的每客户单线程模型，如图 21.2 所示。

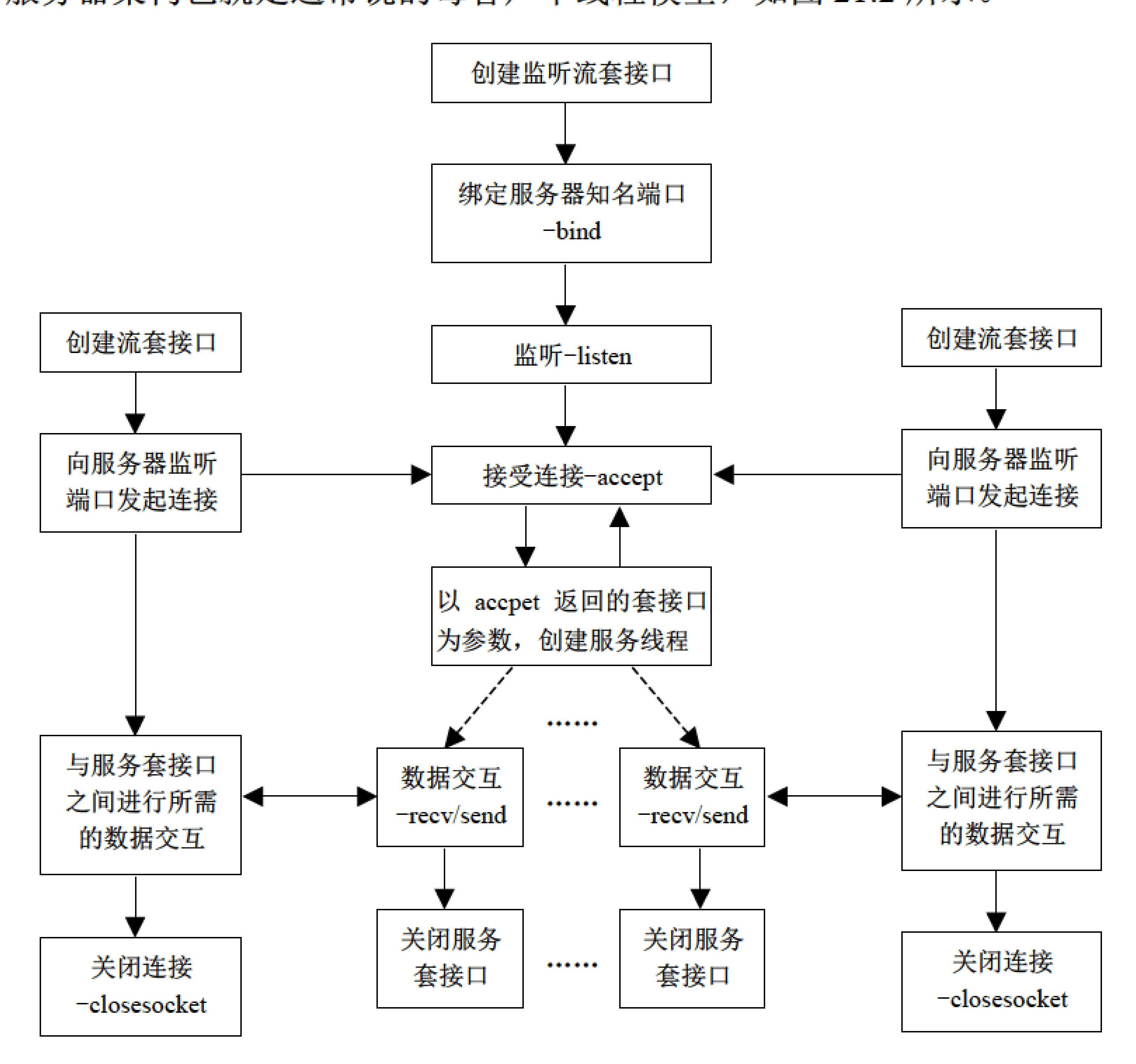


图 21.2　并发 TCP 服务器—每客户单线程模型

基于这种模型的服务器程序流程如下：

（1）创建监听流套接口。

（2）绑定本地的服务器知名端口。

（3）调用 listen 函数，使套接口处于监听状态。

（4）接受外来连接请求，accept 函数返回服务套接口。

（5）以服务套接口为参数，创建服务线程，主线程返回步骤（4）；同时，服务线程进入下一步。

（6）服务线程完成客户端的服务请求，关闭服务套接口并结束线程。

一个简单的 Echo 服务器示例如程序 21.1 所示。在主线程中，服务器循环调用 accept 函数以接受外来客户端连接请求，并将 accept 返回的套接口 sockAccept 作为参数创建服务线程 WorkThread；服务线程在 while(1)循环中接收并回放客户端传来的数据，直到接收操作超时或者某个套接口函数操作失败。

```
************************** 程序21.1 TCPServerA **************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>
#include <PROCESS.H>

void WorkThread(LPVOID lpParam)
{
    SOCKET sockSvr = (SOCKET) lpParam;
    fd_set readSet;
    int ret;
    timeval tv;
    char buf[5000];

    while(1){
        FD_ZERO(&readSet);
        FD_SET(sockSvr, &readSet);
        tv.tv_sec = 5;
        tv.tv_usec = 0;
        ret = select(0, &readSet, NULL, NULL, &tv);
        if(ret == SOCKET_ERROR || ret == 0){// Error or Timeout
            printf("Select error (%d) or Timeout!\n", WSAGetLastError());
            break;
        }
        if(FD_ISSET(sockSvr, &readSet)){
            memset(buf, 0, 5000);
            ret = recv(sockSvr, buf, 5000, 0);
            if(ret == SOCKET_ERROR || ret == 0){// Error or Peer closed
                printf("recv error (%d) or Timeout!\n", WSAGetLastError());
                break;
            }
            //printf("Socket %d recv: %s\n", sockSvr, buf);
            ret = send(sockSvr, buf, strlen(buf), 0);
            if(ret == SOCKET_ERROR)
                break;
```

```
        }
    }

    closesocket(sockSvr);
}

int main(int argc, char* argv[])
{
    WSAData wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);

    SOCKET sockListen = socket(AF_INET, SOCK_STREAM, 0);
    BOOL bReuseAddr = true;
    setsockopt(sockListen, SOL_SOCKET, SO_REUSEADDR, (char *) &bReuseAddr,
    sizeof(bReuseAddr));
    struct sockaddr_in local;
    memset(&local, 0, sizeof(local));
    local.sin_addr.s_addr = INADDR_ANY;
    local.sin_family = AF_INET;
    local.sin_port = htons(9999);
    if(bind(sockListen, (struct sockaddr *) &local, sizeof(local)) ==
    SOCKET_ERROR){
        printf("bind: %d\n", WSAGetLastError());
        closesocket(sockListen);
        WSACleanup();

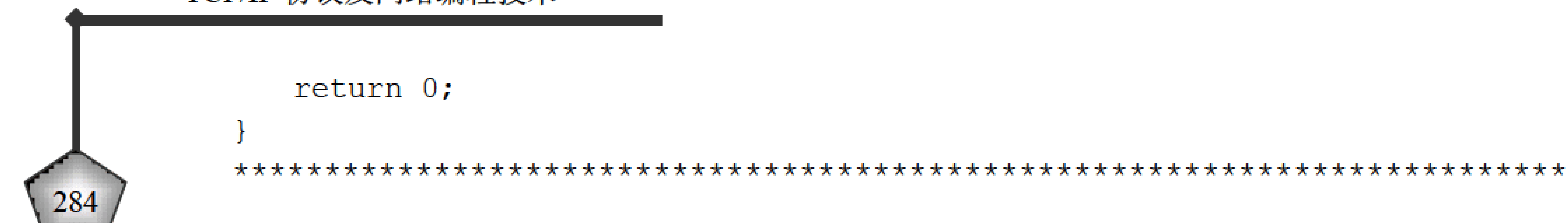
        return -1;
    }

    if(listen(sockListen, 5) == SOCKET_ERROR){
        printf("listen: %d\n", WSAGetLastError());
        closesocket(sockListen);
        WSACleanup();

        return -1;
    }

    SOCKET sockAccept;
    while(true){
        sockAccept = accept(sockListen, NULL, NULL);
        if(sockAccept == INVALID_SOCKET)
            break;
        else
            _beginthread(WorkThread, 0, (LPVOID) sockAccept);
    }

    closesocket(sockListen);
    WSACleanup();
```



```
    return 0;
}
*************************************************************************
```

21.2.2　线程池

事实上，早期的服务器程序采用的都是每客户单进程模型，与进程相比，线程引起的系统负荷无疑要小多了，但是线程的创建也远非零耗费的。因此，在实际的服务器开发中通常采用在应用进程初始化时预先创建多个工作线程即线程池的方法来避免这种开销。

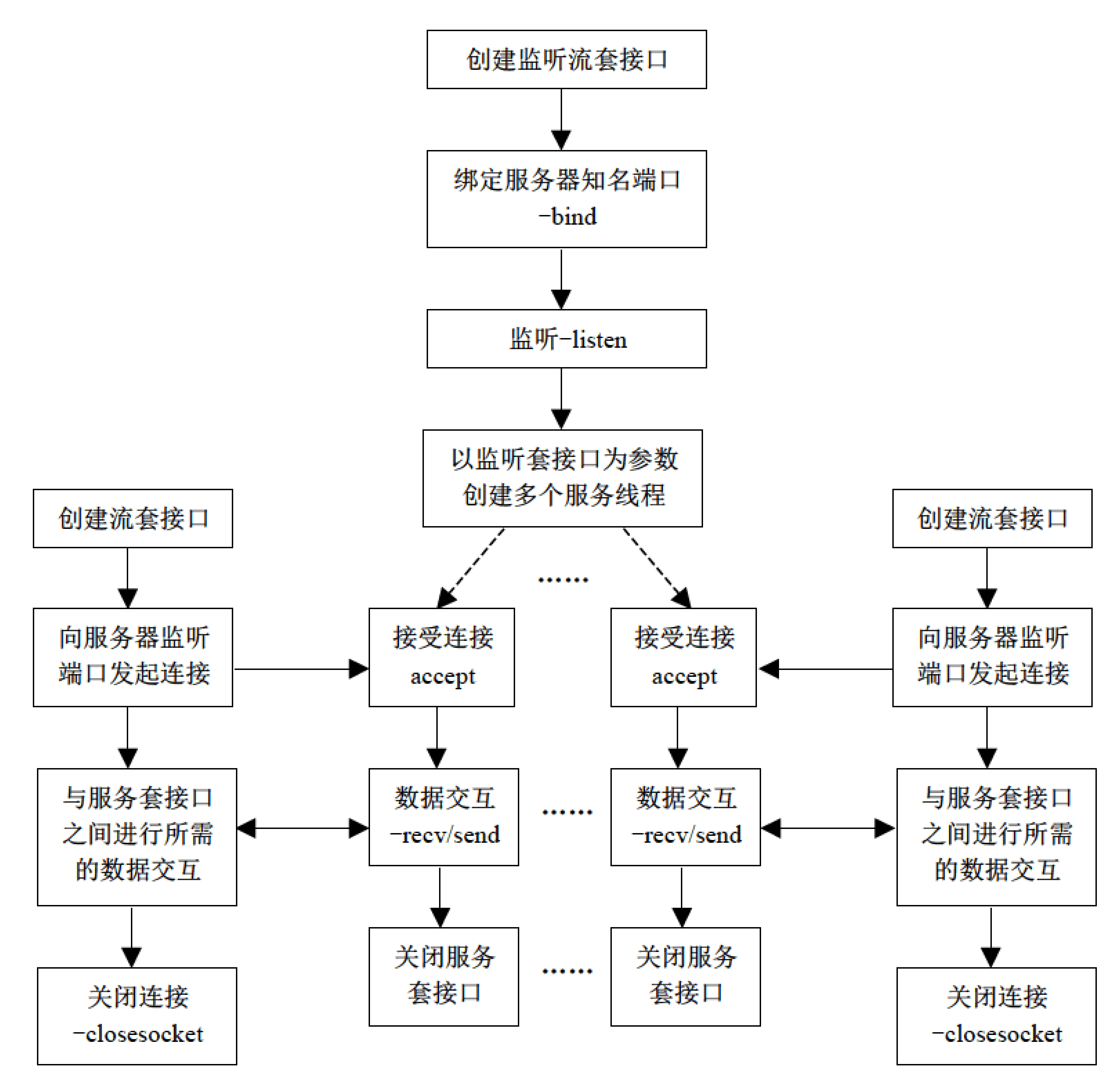


图 21.3　并发 TCP 服务器—线程池模型

一种较为简单的基于线程池模型的服务器程序流程如下：

（1）创建监听流套接口。

（2）绑定本地的服务器知名端口。

（3）调用 listen 函数，使套接口处于监听状态。

（4）以监听套接口为参数创建多个服务线程，主线程休眠。

服务线程的流程如下：

（1）接受外来连接请求，accept 函数返回服务套接口。

（2）处理客户端服务的请求，关闭服务套接口，并返回步骤（1）。

下面的代码演示了基于线程池的 TCP 服务器模型的应用。该程序完成的功能与程序 21.1 基本相同，但从结构上来说有很大的差别。主线程首先创建了监听套接口 sock，然后以 sock 为参数创建了 CO_THREAD_NUM 个服务线程 WorkThread，随后主线程调用 Sleep 进入休眠；服务线程有两个 while(1)循环，其中的外循环接受客户端连接，内循环对接受的连接提供 Echo 服务。

```
*************************** 程序21.2 TCPServerB ***************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>
#include <PROCESS.H>

#define CO_THREAD_NUM  24

void WorkThread(LPVOID lpParam)
{
   SOCKET sockListen = (SOCKET) lpParam;

   SOCKET sockSvr;
   fd_set readSet;
   int ret;
   timeval tv;
   char buf[5000];
   while(1){
       sockSvr = accept(sockListen, NULL, NULL);
       if(sockSvr == INVALID_SOCKET){
           printf("accept: %d\n", WSAGetLastError());
           continue;
       }

       while(1){
           FD_ZERO(&readSet);
           FD_SET(sockSvr, &readSet);
           tv.tv_sec = 5;
           tv.tv_usec = 0;
           ret = select(0, &readSet, NULL, NULL, &tv);
           if(ret == SOCKET_ERROR || ret == 0){// Error or Timeout
              printf("Socket %d: Select error (%d) or Timeout!\n", sockSvr,
              WSAGetLastError());
               break;
           }
           if(FD_ISSET(sockSvr, &readSet)){
               memset(buf, 0, 5000);
               ret = recv(sockSvr, buf, 5000, 0);
               if(ret == SOCKET_ERROR || ret == 0){// Error or Peer closed
                   printf("recv error (%d) or Peer closed!\n", WSAGetLastError());
```

```
                    break;
                }
                //printf("Socket %d recv: %s\n", sockSvr, buf);
                ret = send(sockSvr, buf, strlen(buf), 0);
                if(ret == SOCKET_ERROR)
                    break;
            }
        }

        closesocket(sockSvr);
    }
}

int main(int argc, char* argv[])
{
    WSAData wsaData;
    WSAStartup(WINSOCK_VERSION, &wsaData);

    SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);

    BOOL bReuseAddr = true;
    setsockopt(sock,  SOL_SOCKET,  SO_REUSEADDR,  (char  *)  &bReuseAddr,
    sizeof(bReuseAddr));

    struct sockaddr_in local;
    memset(&local, 0, sizeof(local));
    local.sin_addr.s_addr = INADDR_ANY;
    local.sin_family = AF_INET;
    local.sin_port = htons(9999);
    if(bind(sock, (struct sockaddr *) &local, sizeof(local)) == SOCKET_ERROR){
        printf("bind: %d\n", WSAGetLastError());
        closesocket(sock);
        WSACleanup();

        return -1;
    }

    if(listen(sock, 5) == SOCKET_ERROR){
        printf("listen: %d\n", WSAGetLastError());
        closesocket(sock);
        WSACleanup();

        return -1;
    }

    for(int i = 0; i < CO_THREAD_NUM; i++)
        _beginthread(WorkThread, 0, (LPVOID) sock);
```

```
    Sleep(INFINITE);

    closesocket(sock);
    WSACleanup();
    return 0;
}
```

****

21.2.3　IOCP

线程池模型的缺点是为了能同时满足多个客户请求，不管什么时候服务器系统中都必须有一定数量的线程数在运行，但是由于系统中只能有 CPU 个数那么多个线程同时处于运行态（Running），因此线程池中同时存在很多工作线程并没有实际意义，相反由于几乎所有这些线程均处于可运行态（Runnable），Windows 内核在调度处于运行态的线程时引起的线程上下文的切换会带来很大的资源耗费。为了解决这个问题，并同时保留线程池模型的优势，微软设计了完成端口模型，在 18.2.5 节对模型本身已经进行了深入的分析，本节侧重介绍模型的应用。

IOCP 服务器流程如图 21.4 所示，该模型的使用过程大体如下：

（1）使用 CreateIoCompletionPort 函数创建完成端口，并以该 I/O 完成端口为参数创建多个服务线程。

（2）创建监听套接口。

（3）接受客户端连接请求，返回服务套接口。

（4）将服务套接口与完成端口绑定，并在该套接口上投递初始 I/O 操作请求。

（5）返回步骤（3）。

服务线程的流程如下：

（1）调用 GetQueuedCompletionStatus 函数等待获取完成信息。

（2）根据需要对数据进行处理并投递后续的 I/O 操作请求。

（3）返回步骤（1）。

在给出 IOCP 示例程序之前，先介绍另一组线程操作函数。在本节之前的所有应用程序中都是采用_beginthread 函数来创建线程的，事实上，在 Win32 平台下用的更多的是 CreateThread 函数。该函数的原型如下：

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,   // SD
    DWORD dwStackSize,                          // initial stack size
    LPTHREAD_START_ROUTINE lpStartAddress,      // thread function
    LPVOID lpParameter,                         // thread argument
    DWORD dwCreationFlags,                      // creation option
    LPDWORD lpThreadId                          // thread identifier
);
```

其中 lpThreadAttributes 用于指定线程的安全设置，dwStackSize 是线程栈的初始大小，

dwCreationFlags 用于设定创建线程时的额外参数，lpThreadId 用于返回生成的线程 ID，其余参数与_beginthread 函数一致。因此，忽略多余的参数，下面两行代码的效果是一样的。

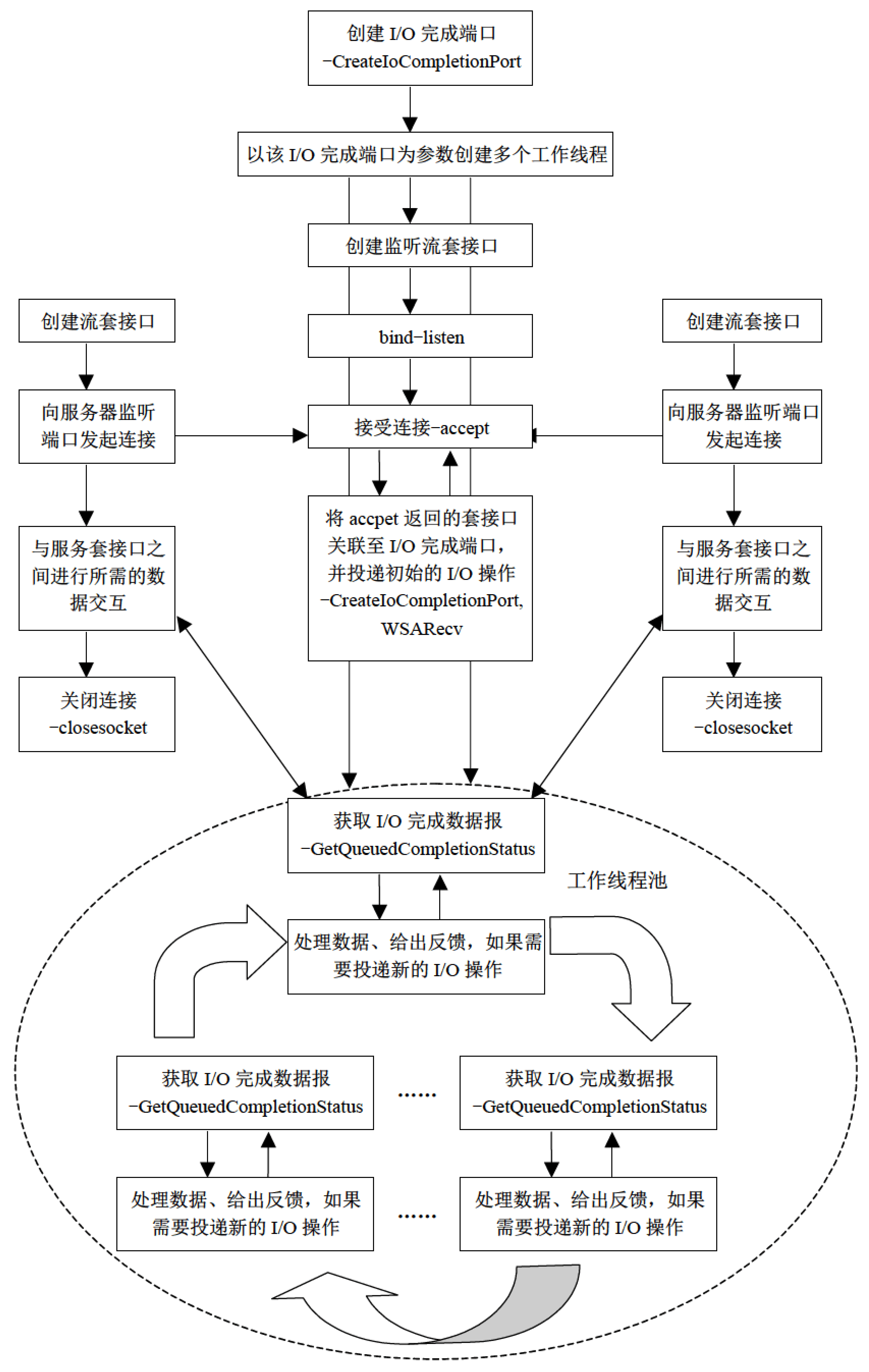


图 21.4　并发 TCP 服务器—IOCP 模型

```
_beginthread(WorkThread, 0, (LPVOID) sock)
CreateThread(NULL, 0, WorkThread, sock, 0, NULL) ;
```

这里需要注意两者的区别。

- 当线程结束时，_beginthread 创建的线程会自动调用 endthread 来释放所有的资源(不能调用 CloseHandle 来关闭句柄)；而 CreateThread 线程必须显式地调用 CloseHandle 来关闭线程的句柄。
- C 运行库里的函数如 printf 函数，必须在由同属于 C 运行库的_begingthread 函数创建的线程中使用；如果在 CreateThread 线程中使用这些 C 运行库函数会引起内存泄漏。

在下面的示例程序中，希望主线程在所有服务线程都结束的情况下才退出，这就需要用到另一个线程同步的函数 WaitForMultipleObjects，函数定义如下：

```
DWORD WaitForMultipleObjects(
  DWORD nCount,               // number of handles in array
  CONST HANDLE *lpHandles,    // object-handle array
  BOOL fWaitAll,              // wait option
  DWORD dwMilliseconds        // time-out interval
);
```

其中的被等待的对象可以是 Change notification、Console input、Event、Job、Mutex、Process、Semaphore、Thread、Waitable timer 等。对于 CreateThread 创建的线程，当它在运行时，状态被置为未触发；当线程结束，状态被置为已触发。因此，假设 hThreadHandles 数组里保存了所有的服务线程句柄，那么主线程与这些线程的同步代码可以是：

```
WaitForMultipleObjects(nThreadCount, hThreadHandles, TRUE, dwMilliseconds)
```

下面是使用完成端口模型的 TCP Echo 服务器程序代码：

```
********************************* 程序 21.3 IOCP ******************************
#pragma comment(lib, "ws2_32.lib")

#include <STDIO.H>
#include <WINSOCK2.H>
#include <PROCESS.H>

#define MAX_THREAD_NUM        24
#define MAX_BUF_LEN           5000

typedef enum _IO_OPER{
    SVR_IO_READ,
    SVR_IO_WRITE
}IO_OPER, *LPIO_OPER;

// 扩展重叠结构体，单 I/O 数据
```

```
typedef struct _OverLappedEx{
      OVERLAPPED    OverLapped;
      WSABUF        wbuf;
      char          data[MAX_BUF_LEN];
      IO_OPER       oper;
      DWORD         flags;
} PER_IO_DATA, *LPPER_IO_DATA;
// 完成键结构体，单句柄数据，对应每个服务套接口一每个连接
typedef struct _CONN_CTX{
      SOCKET        sockAccept;
      LPPER_IO_DATA pPerIOData;
      struct _CONN_CTX *pPrec;
      struct _CONN_CTX *pNext;
} CONN_CTX, *LPCONN_CTX;

CRITICAL_SECTION g_CriticalSection;
LPCONN_CTX g_ptrConnCtxHead = NULL; // 双向链表，用于保存服务器所有连接信息
SOCKET g_sockListen = INVALID_SOCKET;

BOOL WINAPI CtrlHandler(DWORD dwEvent);
// 完成端口操作函数
HANDLE CreateNewIoCompletionPort(DWORD dwNumberOfConcurrentThreads);
BOOL AssociateWithIoCompletionPort(HANDLE hComPort, HANDLE hDevice, DWORD
   dwCompKey);
// 创建监听套接口
SOCKET CreateListenSock();
// 创建连接数据信息
LPCONN_CTX CreateConnCtx(SOCKET sockAccept, HANDLE hIOCP);
void ConnListAdd(LPCONN_CTX lpConnCtx);
void ConnListRemove(LPCONN_CTX lpConnCtx);
void ConnListClear();

int myprintf(const char *lpFormat, ...);
// 工作线程函数
DWORD WINAPI WorkThread(LPVOID lpParam);

int main(int argc, char* argv[])
{
      HANDLE hIOCP = NULL;
      HANDLE hThreadHandles[MAX_THREAD_NUM];
      int nThreadCount = 0;

      WSAData wsaData;
```

```
    if(WSAStartup(WINSOCK_VERSION, &wsaData) != 0){
        myprintf("Winsock initialized failed ...\n");
        return -1;
    }

    for(int i = 0; i < MAX_THREAD_NUM; i++)
        hThreadHandles[i] = NULL;

    if(!SetConsoleCtrlHandler(CtrlHandler, TRUE)){
        myprintf("SetConsoleCtrlHandler: %d\n", GetLastError());
        return -1;
    }

    InitializeCriticalSection(&g_CriticalSection);

    __try{
        // 创建 I/O 完成端口
        hIOCP = CreateNewIoCompletionPort(0);
        if(hIOCP == NULL){
            myprintf("CreateIoCompletionPort: %d\n", GetLastError());
            __leave;
        }

        // 创建多个工作线程
        SYSTEM_INFO sysInfo;
        GetSystemInfo(&sysInfo);
        // 将 sysInfo.dwNumberOfProcessors*2 和 MAX_THREAD_NUM 之间的较小值赋给
           nThreadCount
        nThreadCount = sysInfo.dwNumberOfProcessors * 2 < MAX_THREAD_NUM ?
          (sysInfo.dwNumberOfProcessors * 2) : MAX_THREAD_NUM;
        for(int i = 0; i < nThreadCount; i++){
            HANDLE hThread = CreateThread(NULL, 0, WorkThread, hIOCP, 0, NULL);
            if(hThread == NULL){
                myprintf("CreateThread: %d\n", GetLastError());
                __leave;
            }
            else
                hThreadHandles[i] = hThread;
        }

        g_sockListen = CreateListenSock();
        if(g_sockListen == INVALID_SOCKET)
            __leave;
```

```
        SOCKET sockAccept;
        LPCONN_CTX lpConnCtx;
        int nResult;
        while(true){
            sockAccept = accept(g_sockListen, NULL, NULL);
            if(sockAccept == INVALID_SOCKET)
                __leave;

            lpConnCtx = CreateConnCtx(sockAccept, hIOCP);
            if(lpConnCtx == NULL)
                __leave;
            else
                ConnListAdd(lpConnCtx);

            // 投递初始 I/O 操作
            nResult = WSARecv(sockAccept,
                        &(lpConnCtx->pPerIOData->wbuf),
                        1,
                        NULL,
                        &(lpConnCtx->pPerIOData->flags),
                        &(lpConnCtx->pPerIOData->OverLapped),
                        NULL);
            if((nResult == SOCKET_ERROR) && (WSAGetLastError() != ERROR_IO_
               PENDING)){
                myprintf("WSARecv: %d\n", WSAGetLastError());
                ConnListRemove(lpConnCtx);
                break;
            }
        }
    }
    __finally{
        if(hIOCP){
            for(int i = 0; i < nThreadCount; i++)
                PostQueuedCompletionStatus(hIOCP, 0, 0, NULL);
        }

        // 等待所有工作线程结束
        if( WAIT_OBJECT_0 != WaitForMultipleObjects(nThreadCount,
            hThreadHandles, TRUE, 1000) )
            myprintf("WaitForMultipleObjects failed: %d\n", GetLastError());
        else
            for(int i = 0; i < nThreadCount; i++){
```

```
                if(hThreadHandles[i] != NULL){
                    if(!CloseHandle(hThreadHandles[i]))
                        myprintf("CloseHandle: %d\n", GetLastError());
                }
                hThreadHandles[i] = NULL;
            }

        if(hIOCP){
            CloseHandle(hIOCP);
            hIOCP = NULL;
        }

        if(g_sockListen != INVALID_SOCKET){
            closesocket(g_sockListen);
            g_sockListen = INVALID_SOCKET;
        }
        if(g_ptrConnCtxHead)
            ConnListClear();
    }

    myprintf(".............Stopped.\n");
    DeleteCriticalSection(&g_CriticalSection);
    SetConsoleCtrlHandler(CtrlHandler, FALSE);
    WSACleanup();

    return 0;
}


BOOL WINAPI CtrlHandler(DWORD dwEvent)
{
    SOCKET sockTemp = INVALID_SOCKET;

    switch(dwEvent){
    case CTRL_C_EVENT:
    case CTRL_LOGOFF_EVENT:
    case CTRL_SHUTDOWN_EVENT:
    case CTRL_CLOSE_EVENT:
        myprintf("Server Stopping......\n");
        sockTemp = g_sockListen;
        g_sockListen = INVALID_SOCKET;
        if(sockTemp != INVALID_SOCKET){
            closesocket(sockTemp);
```

```
                sockTemp = INVALID_SOCKET;
            }
            break;
        default:
            return FALSE;
        }

        return TRUE;
}

// 创建 I/O 完成端口
HANDLE CreateNewIoCompletionPort(DWORD dwNumberOfConcurrentThreads){
        return (CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0,
                dwNumberOfConcurrentThreads));
}
// 将套接口与完成端口绑定
BOOL AssociateWithIoCompletionPort(HANDLE hComPort, HANDLE hDevice, DWORD
    dwCompKey){
        return(CreateIoCompletionPort(hDevice, hComPort, dwCompKey, 0) ==
               hComPort);
}
// 创建服务器监听套接口
SOCKET CreateListenSock()
{
        // 创建 WSA_FLAG_OVERLAPPED 属性的套接口
        SOCKET sock = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_
          OVERLAPPED);
        if(sock == INVALID_SOCKET) return sock;

        BOOL bReuseAddr = true;
        if(setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *) &bReuseAddr, sizeof
           (bReuseAddr)) == SOCKET_ERROR){
            myprintf("setsocketopt: %d\n", WSAGetLastError());
            closesocket(sock);
            return INVALID_SOCKET;
        }

        struct sockaddr_in local;
        memset(&local, 0, sizeof(local));
        local.sin_addr.s_addr = INADDR_ANY;
        local.sin_family = AF_INET;
        local.sin_port = htons(9999);
        if(bind(sock, (struct sockaddr *) &local, sizeof(local)) == SOCKET_ERROR){
```

```
        myprintf("bind: %d\n", WSAGetLastError());
        closesocket(sock);
        return INVALID_SOCKET;
    }
    if(listen(sock, 5) == SOCKET_ERROR){
        myprintf("listen: %d\n", WSAGetLastError());
        closesocket(sock);

        return INVALID_SOCKET;
    }

    return sock;
}

LPCONN_CTX CreateConnCtx(SOCKET sockAccept, HANDLE hIOCP)
{
    LPCONN_CTX lpConnCtx = (LPCONN_CTX) GlobalAlloc(GPTR, sizeof(CONN_CTX));
    if(lpConnCtx == NULL) return NULL;
    lpConnCtx->pPerIOData = (LPPER_IO_DATA) GlobalAlloc(GPTR, sizeof(PER_IO_DATA));
    if(lpConnCtx->pPerIOData == NULL){
        GlobalFree(lpConnCtx);
        lpConnCtx = NULL;
        return NULL;
    }

    // 赋值
    lpConnCtx->pNext = NULL;
    lpConnCtx->pPrec = NULL;
    lpConnCtx->sockAccept = sockAccept;

    ZeroMemory(lpConnCtx->pPerIOData, sizeof(PER_IO_DATA));
    lpConnCtx->pPerIOData->OverLapped.hEvent = NULL;
    lpConnCtx->pPerIOData->OverLapped.Internal = 0;
    lpConnCtx->pPerIOData->OverLapped.InternalHigh = 0;
    lpConnCtx->pPerIOData->OverLapped.Offset = 0;
    lpConnCtx->pPerIOData->OverLapped.OffsetHigh = 0;
    lpConnCtx->pPerIOData->wbuf.buf = (char *) lpConnCtx->pPerIOData->data;
    lpConnCtx->pPerIOData->wbuf.len = MAX_BUF_LEN;
    lpConnCtx->pPerIOData->oper = SVR_IO_READ;
    lpConnCtx->pPerIOData->flags = 0;

    // 将套接口与完成端口绑定
```

```
    if(!AssociateWithIoCompletionPort(hIOCP, (HANDLE) sockAccept, (DWORD)
       lpConnCtx)){
        myprintf("AssociateWithIoCompletionPort: %d\n", GetLastError());
        GlobalFree(lpConnCtx->pPerIOData);
        GlobalFree(lpConnCtx);
        lpConnCtx = NULL;
        return NULL;
    }

    return lpConnCtx;
}

void ConnListAdd(LPCONN_CTX lpConnCtx)
{
    LPCONN_CTX    pTemp;

    EnterCriticalSection(&g_CriticalSection);

    if(g_ptrConnCtxHead == NULL) {
        // 链表的第一个（惟一）节点
        lpConnCtx->pPrec = NULL;
        lpConnCtx->pNext = NULL;
        g_ptrConnCtxHead = lpConnCtx;
    }else{
        // 加到链表头部
        pTemp = g_ptrConnCtxHead;
        g_ptrConnCtxHead = lpConnCtx;
        lpConnCtx->pNext = pTemp;
        lpConnCtx->pPrec = NULL;
        pTemp->pPrec = lpConnCtx;
    }

    LeaveCriticalSection(&g_CriticalSection);
}

void ConnListRemove(LPCONN_CTX lpConnCtx)
{
    LPCONN_CTX pPrec;
    LPCONN_CTX pNext;

    EnterCriticalSection(&g_CriticalSection);
    if(lpConnCtx) {
        pPrec = lpConnCtx->pPrec;
```

```
        pNext = lpConnCtx->pNext;
        if((pPrec == NULL) && (pNext == NULL)){// [*] -> NULL：链表惟一节点
            g_ptrConnCtxHead = NULL;
        }
        else if((pPrec == NULL) && (pNext != NULL)){// [*] -> [ ] -> .... [ ]：链表首节点
            pNext->pPrec = NULL;
            g_ptrConnCtxHead = pNext;
        }
        else if((pPrec != NULL) && (pNext == NULL)){// [ ] -> [ ] -> .... [*]：链表末节点
            pPrec->pNext = NULL;
        }
        else if( pPrec && pNext ) {// [ ] -> [*] -> .... [ ]：链表中间节点
            pPrec->pNext = pNext;
            pNext->pPrec = pPrec;
        }

        // 关闭连接，释放资源
        closesocket(lpConnCtx->sockAccept);
        GlobalFree(lpConnCtx->pPerIOData);
        GlobalFree(lpConnCtx);
        lpConnCtx = NULL;
    }

    LeaveCriticalSection(&g_CriticalSection);
    return;
}

void ConnListClear()
{
    LPCONN_CTX pTemp1, pTemp2;

    EnterCriticalSection(&g_CriticalSection);
    pTemp1 = g_ptrConnCtxHead;
    while(pTemp1) {
        pTemp2 = pTemp1->pNext;
        ConnListRemove(pTemp1);
        pTemp1 = pTemp2;
    }
    LeaveCriticalSection(&g_CriticalSection);
    return;
}
```

```
int myprintf(const char *lpFormat, ...)
{
    int nLen = 0;
    int nRet = 0;
    char cBuffer[512] ;
    va_list arglist;
    HANDLE hOut = NULL;

    ZeroMemory(cBuffer, sizeof(cBuffer));

    va_start(arglist, lpFormat);

    nLen = lstrlen( lpFormat ) ;
    nRet = wvsprintf(cBuffer, lpFormat, arglist);

    if(nRet >= nLen || GetLastError() == 0) {
        hOut = GetStdHandle(STD_OUTPUT_HANDLE) ;
        if(hOut != INVALID_HANDLE_VALUE)
            WriteConsole(hOut, cBuffer, lstrlen(cBuffer), (LPDWORD)&nLen,
               NULL);
    }

    return nLen;
}

DWORD WINAPI WorkThread(LPVOID lpParam)
{
    HANDLE hIOCP = (HANDLE) lpParam;
    BOOL bSuccess = false;
    DWORD     dwIOSize;
    LPPER_IO_DATA lpPerIOData;
    LPOVERLAPPED pOverLapped;
    LPCONN_CTX lpConnCtx;
    int nResult;

    while(1){
        bSuccess = GetQueuedCompletionStatus(hIOCP, &dwIOSize, (LPDWORD)
           &lpConnCtx, &pOverLapped, INFINITE);
        if(!bSuccess)
            myprintf("%d\n", GetLastError());
        if(lpConnCtx == NULL)
            return 1;
```

```
        lpPerIOData = (LPPER_IO_DATA) (pOverLapped);
        if(!bSuccess || (bSuccess && (dwIOSize == 0))){
            ConnListRemove(lpConnCtx);
            continue;
        }
#ifdef _DEBUG
        myprintf("Different way to obtain PER_IO_DATA\n");
        myprintf("The two one must be equal - A:%x\tB:%x\n",
                    lpConnCtx->pPerIOData,
                    lpPerIOData);
#endif
        switch(lpPerIOData->oper){
        case SVR_IO_WRITE:// send then recv
#ifdef _DEBUG
            myprintf("Socket %d send: %s\n", lpConnCtx->sockAccept,
                    lpPerIOData->wbuf.buf);
#endif
            ZeroMemory(lpPerIOData, sizeof(PER_IO_DATA));
            lpPerIOData->OverLapped.hEvent = NULL;
            lpPerIOData->OverLapped.Internal = 0;
            lpPerIOData->OverLapped.InternalHigh = 0;
            lpPerIOData->OverLapped.Offset = 0;
            lpPerIOData->OverLapped.OffsetHigh = 0;
            lpPerIOData->wbuf.buf = (char *) &(lpPerIOData->data);
            lpPerIOData->wbuf.len = MAX_BUF_LEN;
            lpPerIOData->oper = SVR_IO_READ;
            lpPerIOData->flags = 0;

            nResult = WSARecv(lpConnCtx->sockAccept,
                            &(lpPerIOData->wbuf),
                            1,
                            NULL,
                            &(lpPerIOData->flags),
                            &(lpPerIOData->OverLapped),
                            NULL);
            if(nResult == SOCKET_ERROR && WSAGetLastError() != ERROR_IO_
               PENDING){
                myprintf("WSARecv: %d\n", WSAGetLastError());
                ConnListRemove(lpConnCtx);
            }
            break;
        case SVR_IO_READ:// recv then echo
#ifdef _DEBUG
```

```
                myprintf("Socket %d recv: %s\n", lpConnCtx->sockAccept,
                   lpPerIOData->wbuf.buf);
#endif
                lpPerIOData->wbuf.len = dwIOSize;
                lpPerIOData->oper = SVR_IO_WRITE;
                lpPerIOData->flags = 0;

                nResult = WSASend(lpConnCtx->sockAccept,
                                &(lpPerIOData->wbuf),
                                1,
                                NULL,
                                lpPerIOData->flags,
                                &(lpPerIOData->OverLapped),
                                NULL);
                if(nResult == SOCKET_ERROR && WSAGetLastError() != ERROR_IO_
                   PENDING){
                        myprintf("WSASend: %d\n", WSAGetLastError());
                        ConnListRemove(lpConnCtx);
                }
                break;
            default:;
            }
        }

        return 0;
}
```

****

程序 21.3 给出了基于 IOCP 的 ECHO 服务器代码，该程序完成的功能与程序 21.2、21.1 完全相同。下面对它进行详细的分析介绍。

1．数据结构和全局变量

❒ IO_OPER 数据结构

```
typedef enum _IO_OPER{
    SVR_IO_READ,
    SVR_IO_WRITE
}IO_OPER, *LPIO_OPER;
```

枚举类型，用于指示服务器 I/O 操作的类型。

❒ PER_IO_DATA 数据结构

```
typedef struct _OverLappedEx{
    OVERLAPPED            OverLapped;
    WSABUF                wbuf;
```

```
    Char                    data[MAX_BUF_LEN];
    IO_OPER                 oper;
    DWORD                   flags;
} PER_IO_DATA, *LPPER_IO_DATA;
```

该结构体是扩展重叠结构，单 I/O 数据。其中 wbuf 是 I/O 操作的数据对象；data 是实际的数据缓冲区；oper 用于标志 I/O 操作的类型，IO_OPER 枚举型，可以是 SVR_IO_READ 或者是 SVR_IO_WRITE；flags 用于设定或者返回 I/O 操作的标志。

❒ CONN_CTX 数据结构

```
typedef struct _CONN_CTX{
    SOCKET                  sockAccept;
    LPPER_IO_DATA           pPerIOData;
    struct _CONN_CTX        *pPrec;
    struct _CONN_CTX        *pNext;
} CONN_CTX, *LPCONN_CTX;
```

单句柄数据结构，用于保存每个客户端的连接信息。其中 sockAccept 是该连接的服务器端服务套接口；pPerIOData 指向该连接的 I/O 操作信息；pPrex 和 pNext 用于形成服务器当前所有连接信息的双向链表，分别指向链表中的前一个节点和后一个节点。

❒ g_CriticalSection 全局变量

用于主线程和各个服务线程之间的同步，主要目的是防止对连接信息链表的访问冲突。

❒ g_ptrConnCtxHead 全局变量

指向连接信息双向链表的首节点（最新加入的客户端连接），用于对该链表的访问和维护。

2．函数

❒ BOOL WINAPI CtrlHandler(DWORD dwEvent)

该函数对控制台消息进行处理，当接收到 CTRL_C_EVENT、CTRL_LOGOFF_EVENT、CTRL_SHUTDOWN_EVENT 或者 CTRL_CLOSE_EVENT 事件时，服务器将关闭监听套接口，从而导致主线程从接收连接的死循环中退出，并最终结束所有服务线程、释放连接并停机。

❒ LPCONN_CTX CreateConnCtx(SOCKET sockAccept, HANDLE hIOCP)

当服务器接受了客户端连接请求后，将返回的服务套接口和完成端口作为参数调用该函数。该函数完成服务套接口与完成端口的绑定，以及为该连接的相关信息分配存储区的工作。

❒ void ConnListAdd(LPCONN_CTX lpConnCtx)

本函数将新的连接信息加入到全局的连接信息链表。

❒ void ConnListRemove(LPCONN_CTX lpConnCtx)

本函数将指定的连接信息从全局连接信息链表中删去，并关闭连接、释放相应的存储区资源。

❒ void ConnListClear()

本函数完成服务器退出时关闭连接、释放资源的工作。对全局连接信息链表中的每个节点，逐个调用 ConnListRemove 函数。

❒ int myprintf(const char *lpFormat, ...)

由于 printf 函数只能在用 C 运行库中函数创建的线程中使用，本程序重写了自己的输出函数。

3．流程

整个服务器的流程与图 21.4 介绍的基本一致，惟一的差别在于服务器的退出处理。当服务器控制台接收到特定的退出消息后，消息处理函数将监听套接口关闭，从而导致主线程退出服务循环；随后，主线程调用 PostQueuedCompletionStatus 函数向所有的服务线程发送完成键为 NULL 的完成信息，通知各服务线程退出服务；在所有服务线程均结束后，主线程对全局连接信息链表中的连接逐个关闭并释放所用资源；在资源释放完毕后，主线程结束　运行。

4．代码分析

第 5～6 行定义了两个常量 MAX_THREAD_NUM 和 MAX_BUF_LEN，分别用于表示最大的服务线程数目和服务器 I/O 缓冲区的大小。

第 7～25 行定义了所需使用的 3 个数据结构。

第 26～28 行定义了 3 个全局变量 g_CriticalSection、g_ptrConnCtxHead 和 g_sockListen。

第 29～42 行声明了若干函数。

第 43～146 行 main 函数。

- ❒ 48～52 行，初始化 winsock。
- ❒ 55～58 行，调用 SetConsoleCtrlHandler 函数设置控制台事件的响应函数。
- ❒ 59 行，初始化临界区变量 g_CriticalSection。
- ❒ 60～111 行，__try 程序段。这部分代码使用了 C++的 try-finally 程序结构。如果在__try 程序段代码执行过程中出现了异常或者调用了__leave 语句，那么程序将跳转至__finally 程序段执行。
  - ✧ 61～66 行，创建 I/O 完成端口 hIOCP。
  - ✧ 68～80 行，将 sysInfo.dwNumberOfProcessors*2 和 MAX_THREAD_NUM 之间的较小值赋给 nThreadCount；然后创建 nThreadCount 个工作线程，并将线程句柄保存在 hThreadHandles 数组中。
  - ✧ 81～83 行，调用 CreateListenSock 函数创建监听套接口 g_sockListen。
  - ✧ 87～110 行，循环体。服务器循环接受外来连接请求，创建连接信息 lpConnCtx 并保存至全局连接信息链表，最后投递初始 WSARecv 操作请求。
- ❒ 112～139 行，__finally 程序段，进行资源释放等程序结束的清理工作。
  - ✧ 113～116 行，如果完成端口句柄 hIOCP 不为 NULL，投递 nThreadCount 个完成键为 NULL 的完成信息包，通知工作线程终止服务。
  - ✧ 117～127 行，等待工作线程结束，并关闭其句柄。
  - ✧ 128～131 行，关闭完成端口。
  - ✧ 133～136 行，关闭监听套接口。
  - ✧ 137～138 行，调用 ConnListClear 函数清除连接信息链表。
- ❒ 142～145 行，删除临界区变量 g_CriticalSection，取消控制台事件响应函数的设定，并结束 winsock 的使用。

第 147～168 行 CtrlHandler 函数定义。当控制台收到 CTRL_C_EVENT、CTRL_

LOGOFF_EVENT、CTRL_SHUTDOWN_EVENT 或者 CTRL_CLOSE_EVENT 事件时，关闭套接口 g_sockListen。该操作将导致 main 函数中 accept 操作失败，从而跳转至__finally 程序段。

第 169～176 行 CreateNewIoCompletionPort 和 AssociateWithIoCompletionPort 函数定义。

第 177～206 行 CreateListenSock 函数，创建服务器监听套接口。首先创建 WSA_FLAG_OVERLAPPED 属性的套接口，然后在该套接口上启用 SO_REUSEADDR 选项，最后绑定本地 9999 端口并调用 listen 函数使之处于监听状态。

第 207～241 行 CreateConnCtx 函数。为 sockAccept 对应的已接受的客户端连接创建连接信息数据，并将 sockAccept 与完成端口绑定。

第 242～260 行 ConnListAdd 函数，将连接信息 lpConnCtx 加入到全局的连接信息链表中，线程之间对该链表的互斥访问使用临界区变量 g_CriticalSection 来实现。

第 261～293 行 ConnListRemove 函数，将连接信息 lpConnCtx 从全局的连接信息链表中删除，关闭相应的套接口并释放资源。

第 294～306 行 ConnListClear 函数，调用 ConnListRemove 函数清除连接信息链表。

第 307～324 行 myprintf 函数，自定义的控制台输出函数，惟一需要注意的是该函数接收不定长输入变量。

第 325～402 行 WorkThread 线程函数。335～400 行是函数主服务循环。

- 336～351 行，调用 GetQueuedCompletionStatus 函数获取完成信息。如果得到的完成键为空，则终止该工作线程；如果 GetQueuedCompletionStatus 返回 FALSE 或者 I/O 数据量为 0（说明客户端断开了连接），则调用 ConnListRemove 函数清除该连接。
- 352～399 行，根据完成信息中的 lpPerIOData->oper 数据，判断该完成信息所对应的服务器 I/O 操作类型：如果是 SVR_IO_WRITE，那么投递下一次 WSARecv 请求；如果是 SVR_IO_READ，说明已读取了客户端发送的数据，那么投递 WSASend 操作请求，对客户端进行 Echo。

21.3　几种服务器架构的分析与比较

至此，已经介绍完常见的 TCP 服务器架构（如图 21.5 所示），本节对这些架构的优缺点进行简单的分析。

首先是迭代服务器和并发服务器，这两者之间的区别是非常明显的。由于迭代服务器每次只能接收一个连接请求为一个客户端服务，因此这种服务器只适用于一些非常简单的服务类型。通常所看到的 TCP 服务器都是采用并发工作方式，而迭代方式一般用于 UDP 服务器。

其次是并发服务器的 3 种架构，这 3 种架构在实际环境中都能够看到，虽然公认完成端口模型是 Win32 平台下的扩展性最好的服务器模型，但是并不是所有情况下都使用该模型。这是因为一方面完成端口模型相对而言较为复杂；另一方面在服务器负载不大的情况下，另两种模型反而能获得更好的服务性能。针对这 3 种服务器架构的选择问题，给出如下结论：

- 当服务器负载要求较轻时，普通的每客户单线程模型就足够了。服务器启动一个监

听线程，然后为每一个到来的连接请求派生一个工作线程。这些工作线程都是在需要提供服务时创建，服务完毕时消亡。

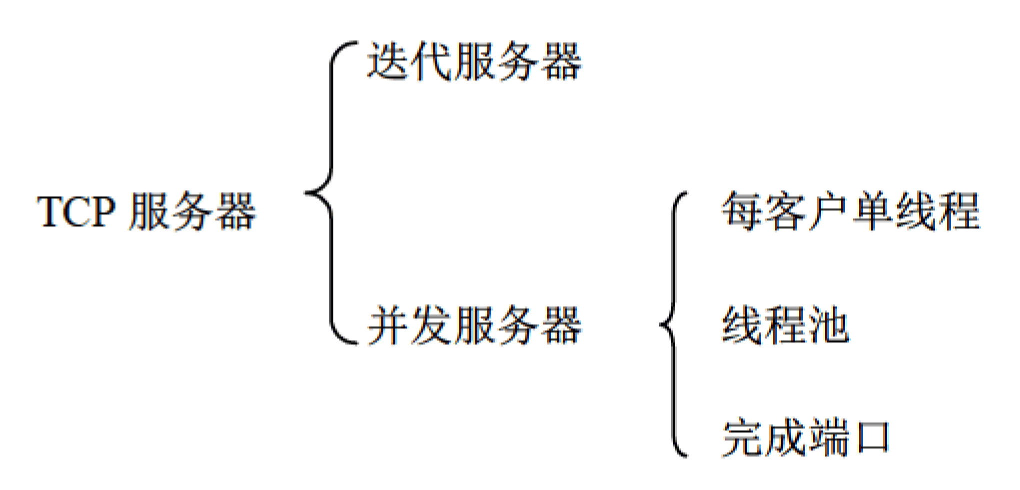


图 21.5　常见 TCP 服务器系统架构

- 相对于为每个到来的客户创建工作线程的方式，预先创建一个线程池可以大大减少系统的耗费。但是，对于实用的系统来说，必须要有一个启发式算法合理地调整线程池中的线程数目，这个算法必须综合考虑当前运行线程数、空闲线程数以及系统负荷等多方面的因素。
- 如果服务器的性能要求很高，需要同时为成千上万的并发客户服务，并且希望系统能充分利用多个处理器的优势，那么完成端口是必然的选择。它不仅具有线程池模型的优点，而且能更充分、高效地利用每个工作线程。同样，动态调整服务器系统中的线程数的智能算法也是必需的。

第 22 章　Internet 编程示例

在前几章用了很大的篇幅介绍如何使用套接口 API 进行网络编程，以及一些常见的 C/S 服务器模型设计和相应的编程方法，本章将用一个完整的 Web 服务器的例子——MyWeb 项目来对这些内容进行演示。

MyWeb 项目应用 Winsock 接口函数实现基本的 WWW 服务器功能，整个系统基于 I/O 完成端口模型，多线程、多用户，可以作为大型软件系统中的 WEB 页面发布模块使用，也可以扩展为一个独立的 Win32 平台下的高性能 Web 服务器。

MyWeb 目前仅支持 HTTP-GET 指令和以 HTM/HTML 为文件后缀名的标准网页。部分代码参考了 Souren Abeghyan 发布在 www.codeproject.com 的“Multithreaded server class with example of HTTP server”一文，在此表示感谢。

22.1　MyWeb 服务器的使用

在给出 MyWeb 的设计与实现的详细分析之前，首先简单地介绍该系统的用户界面以及操作流程。

22.1.1　用户界面

MyWeb 的用户界面可以分为两部分：主对话框（图 22.1）和任务栏图标（图 22.2）。

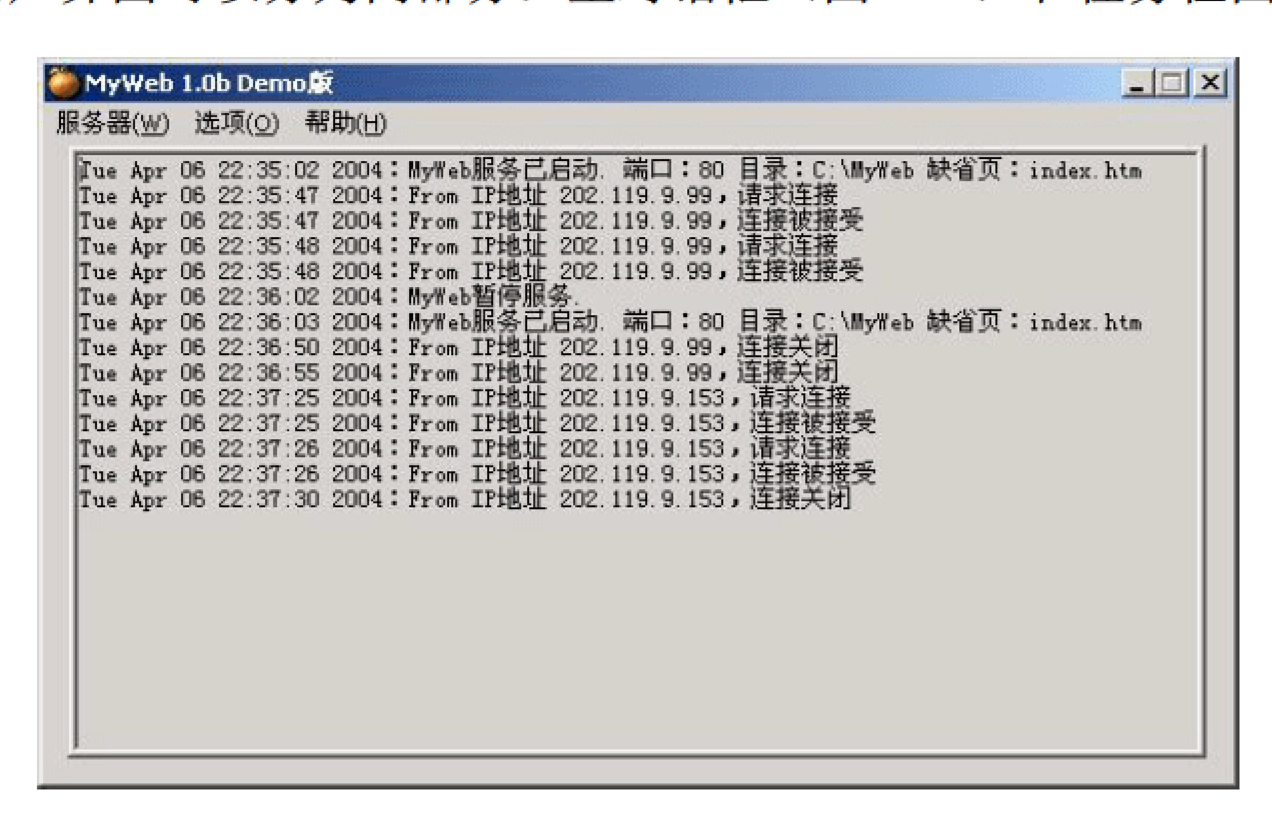


图 22.1　MyWeb 服务器的操作界面

在主对话框中，提供了 3 个菜单项，分别是“服务器”、“选项”和“帮助”。其中“服务器”菜单包含了服务器的 4 个基本操作：“启动”、“暂停”、“停止”和“退出”；“选项”菜单包含两项功能：“配置”服务器与“复位”选项数据，用户可配置的选项数据有

MyWeb 的服务端口（默认为 80）、默认页面（index.htm）以及服务器的工作目录；“帮助”菜单中的“关于”给出了 MyWeb 服务器的一些基本信息。

MyWeb 提供的另一个操作界面是任务栏图标。该图标主要起两方面的作用：① 替代主对话框提供基本的服务器操作界面；② 通过不同图标的切换，显示服务器的当前状态。如图 22.2 所示。

当用户右键单击该图标时，会弹出一个快捷菜单，如图 22.3 所示。该菜单基本上涵盖了主对话框中提供的所有操作功能。



图 22.2　MyWeb 的任务栏图标



图 22.3　MyWeb 弹出菜单

22.1.2　操作流程

MyWeb 的操作非常简单，主要分为两部分：配置系统和启动/暂停/停止服务。

（1）配置系统

在默认情况下，MyWeb 服务器在 80 端口启动 WWW 服务，工作目录为 C:\MyWeb，默认主页是 index.htm。如果这些选项不符合用户的实际情况，那么可以在启动 MyWeb 服务器之前进行系统配置。单击主对话框中的菜单“选项”→“配置”命令，或者单击任务栏图标的弹出菜单中的“配置服务器”命令，系统都将弹出“服务器配置对话框”，如图 22.4 所示。在该对话框中，用户可以设置服务器的基本选项。

如果单击了主菜单中的“选项”→“复位”命令，在得到确认后用户的所有配置信息都将丢失，系统将使用默认配置。



图 22.4　服务器配置

（2）启动/暂停/停止服务

“服务器”菜单的前三项分别用于服务器的启动、暂停和停止，相对应，弹出菜单中有“启动 Myweb”、“暂停 Myweb”和“停止 Myweb”3 个菜单项。当单击“启动”后，MyWeb 将读取系统的配置数据并启动 WWW 服务。如果启动成功，那么在主对话框的系统信息输出框中将给出启动信息，如图 22.1 中间部分所示。此外，服务器的其他操作如暂停、停止，以及客户端的访问情况也会在这里被输出。

此时，打开浏览器在地址栏输入 MyWeb 所在主机的 IP 或者域名，即可访问到相应的主页，如图 22.5 所示。

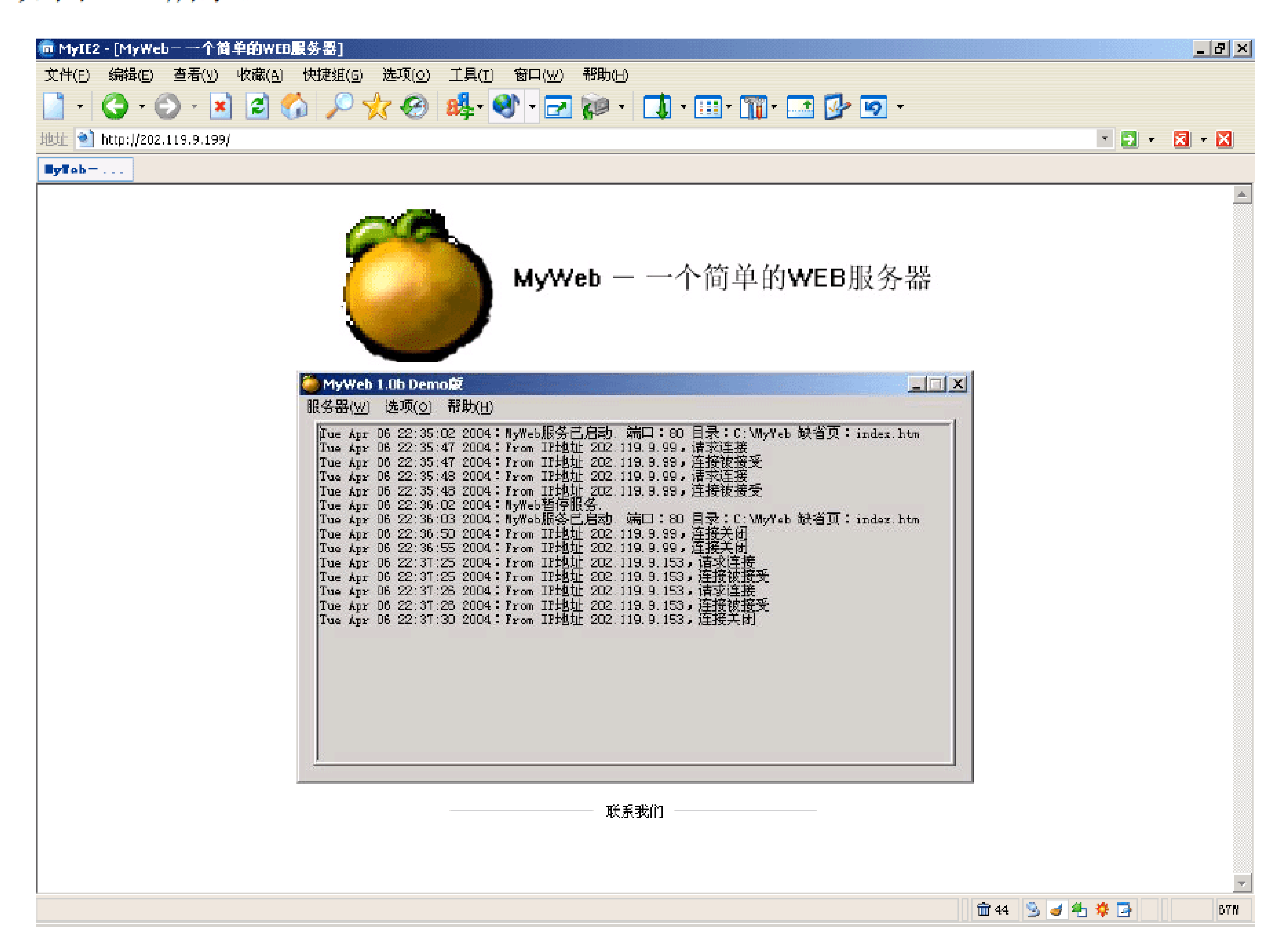


图 22.5 浏览页面

将服务器暂停的作用是，暂时拒绝所有外来的新的连接请求，但是并不关闭已有的连接，因此也不会释放服务器用于保存连接信息的系统资源。如果希望将服务器彻底重启，那么可以选择停止服务器，然后再启动。

22.2 源码及其分析

整个项目采用微软的 MFC 架构，属于基于对话框的应用程序。全部代码除了 VC 自动生成的 CMyWebServerDlg、CMyWebServerApp 和 CaboutDlg 3 个基本类之外，还包含了 COptions、COptionsDlg、CMyNotifyIcon、ChttpServer 4 个类，以及 CONN_CTX、PER_IO_DATA 两个重要数据结构。下面按照先外围后核心、先简单后复杂的顺序对这些类加以分析。

在介绍此部分内容时，假设读者已经掌握基本的 Winsock API 和多线程编程，对 C++的标准模板库 STL 和 VC 的 MFC 框架的使用也有一定的了解，能够独立开发简单的基于对话框的 MFC 应用程序。

22.2.1 COptions 类

COptions 类用于读取/保存服务器的配置参数，所有操作均围绕注册表进行。下面首先分

析 COptions 类的接口声明，源码如下：

```
///////////////////////////////////////////////////////////////////////////////
// Options.h: COptions 类的接口说明. Create: 2004-03-24    Last Modify: 2004-03-24
///////////////////////////////////////////////////////////////////////////////
1   #if !defined(AFX_OPTIONS_H__5A89F18B_E7D1_41BE_B74A_EB3A35D65104__INCLUDED_)
2   #define AFX_OPTIONS_H__5A89F18B_E7D1_41BE_B74A_EB3A35D65104__INCLUDED_

3   #if _MSC_VER > 1000
4   #pragma once
5   #endif // _MSC_VER > 1000

6   class COptions
7   {
8   public:
9       COptions();
10      virtual ~COptions();
11  public:
12      void LoadDefOptions();
13      char m_szDefaultPage[255];
14      char m_szHomeDir[255];
15      UINT m_nServerPort;
16  protected:
17      void SaveOptions();
18      void GetOptions();
19  };

20  extern COptions  *g_pSvrOptions;

21  #endif
///////////////////////////////////////////////////////////////////////////////
```

第 12 行声明了 COptions 类的 public 成员函数 LoadDefOptions，用户可调用该函数复位服务器选项参数。

第 13～15 行声明了 3 个 public 成员变量 m_szDefaultPage、m_szHomeDir 和 m_nServerPort，分别对应 MyWeb 服务器的默认主页、用户目录和服务端口。用户使用时，可以直接对这几个变量进行修改、赋值。

第 17 行 SaveOptions 成员函数，protected 类型，用于将选项参数保存到注册表中。

第 18 行 GetOptions 成员函数，protected 类型，用于从注册表读取选项参数。

第 20 行 g_pSvrOptions，全局变量，指向 COptions 类的指针，该变量由 extern 描述符修饰，外部连接，在 Options.cpp 文件中定义实现。

下面是 COptions 类实现的源码：

```
///////////////////////////////////////////////////////////////////////////////
// Options.cpp: COptions 类的实现, 完成了基于注册表的配置信息读/写功能
```

```
// Create: 2004-03-24      Last Modify: 2004-03-25
///////////////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "MyWebServer.h"
#include "Options.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

COptions _SvrOptions;
COptions *g_pSvrOptions = &_SvrOptions;

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

COptions::COptions()
{
    GetOptions();
}

COptions::~COptions()
{
    SaveOptions();
}

void COptions::GetOptions()
{
    LoadDefOptions();
    HKEYhKey;
    DWORD        dwType, dwcbData;
    RegCreateKey(HKEY_CURRENT_USER,
                    "SOFTWARE\\CSE528.SEU\\MyWebServer\\Options", &hKey);
    dwcbData = 255;
    RegQueryValueEx(hKey, "HomeDir", NULL, &dwType, (LPBYTE) m_szHomeDir,
                &dwcbData);
    dwcbData = 255;
    RegQueryValueEx(hKey, "DefaultPage", NULL, &dwType, (LPBYTE)m_szDefaultPage,
                &dwcbData);
    dwcbData = 255;
    RegQueryValueEx(hKey, "ServerPort", NULL, &dwType, (LPBYTE) &m_nServerPort,
                &dwcbData);
    RegCloseKey(hKey);
```

```
39  }

40  void COptions::SaveOptions()
41  {
42      HKEYhKey;
43      RegOpenKeyEx(HKEY_CURRENT_USER,
44        "SOFTWARE\\CSE528.SEU\\MyWebServer\\Options", 0, KEY_ALL_ACCESS, &hKey);
45      RegSetValueEx(hKey, "HomeDir", NULL, REG_SZ, (CONST BYTE *) m_szHomeDir,
46                  strlen(m_szHomeDir));
47      RegSetValueEx(hKey, "DefaultPage", NULL, REG_SZ, (CONST BYTE *) m_
        szDefaultPage,
48                  strlen(m_szDefaultPage));
49      RegSetValueEx(hKey, "ServerPort", NULL, REG_DWORD,
50                  (CONST BYTE *)&m_nServerPort, sizeof(m_nServerPort));
51      RegCloseKey(hKey);
52  }

53  void COptions::LoadDefOptions()
54  {
55      strncpy(m_szHomeDir, "C:\\MyWeb", 255);
56      strncpy(m_szDefaultPage, "index.htm", 255);
57      m_nServerPort = 80;
58  }
//////////////////////////////////////////////////////////////////////////////
```

第 9～10 行定义了一个 COptions 类的实例_SvrOptions，以及指向该变量的指针 g_pSvrOptions。

第 14～17 行 COptions 类的构造函数，该函数调用类的 protected 成员函数 GetOptions，从注册表中读取数据。

第 18～21 行 COptions 类的析构函数，该函数调用类的 protected 成员函数 SaveOptions，将数据保存至注册表。

第 22～39 行 GetOptions 成员函数，从注册表中读取服务器选项数据，并赋给 3 个 public 成员变量 m_szDefaultPage、m_szHomeDir 和 m_nServerPort。

- 在第 24 行，调用 LoadDefOptions 函数，给 3 个成员变量 m_szDefaultPage、m_szHomeDir 和 m_nServerPort 赋默认值。
- 25～28 行，调用 RegOpenKeyEx 函数读取或者创建注册表项 HKEY_CURRENT_USER\\SOFTWARE\\CSE528.SEU\\MyWebServer\\Options。
- 29～37 行，调用 RegQueryValueEx 函数读取注册表项 HKEY_CURRENT_USER\\SOFTWARE\\CSE528.SEU\\MyWebServer\\Options 的 3 个子项 HomeDir、DefaultPage 和 ServerPort。如果某个子项不存在，则 RegQueryValueEx 函数会创建该子项并将相应的成员变量的当前值（即默认值，见 24 行）写入注册表。
- 38 行，关闭注册表。

第 40～52 行 SaveOptions 成员函数。调用 RegSetValueEx 函数将注册表 m_szDefaultPage、

m_szHomeDir 和 m_nServerPort 的值保存至注册表对应表项。

第 53～58 行给 3 个成员变量 m_szDefaultPage、m_szHomeDir 和 m_nServerPort 赋默认值。

我们在 Options.cpp 文件中的第 9～10 行定义了 COptions 类的实例_SvrOptions，并且将 g_pSvrOptions 赋值为该实例的地址。其中的_SvrOptions 仅在 COptions 模块中可见，而 g_pSvrOptions 为全局变量，可由其他模块通过包含 Options.h 文件来使用。这样设计的意义在于，当 COptions 模块被包含在某个项目中时，由于_SvrOptions 的存在，系统会立即调用 COptions 类的构造函数，从而自动读取注册表中的相关数据；而用户可以通过 g_pSvrOptions 指针来读取或者修改_SvrOptions 的成员变量值；当软件运行结束系统释放资源时，_SvrOptions 的析构函数又会被自动执行，从而将当前的 m_szDefaultPage、m_szHomeDir 和 m_nServerPort 值保存至注册表。

22.2.2　COptSetupDlg 类

COptSetupDlg 类由系统为服务器配置对话框 IDD_DIALOG_OPTSETUP 自动生成，主要向用户提供参数设置的用户界面，并将这些数据赋值给全局的 COptions 类实例变量。

对应于图 22.6 中的输入控件，为该类添加了 3 个成员变量：

CString	m_strDefaultPage;	对应于缺省页面：IDC_COMBO_DEFPAGE
CString	m_strHomeDir;	对应于用户目录：IDC_EDIT_HOMEDIR
UINT	m_nSvrPort;	对应于服务端口：IDC_EDIT_PORT

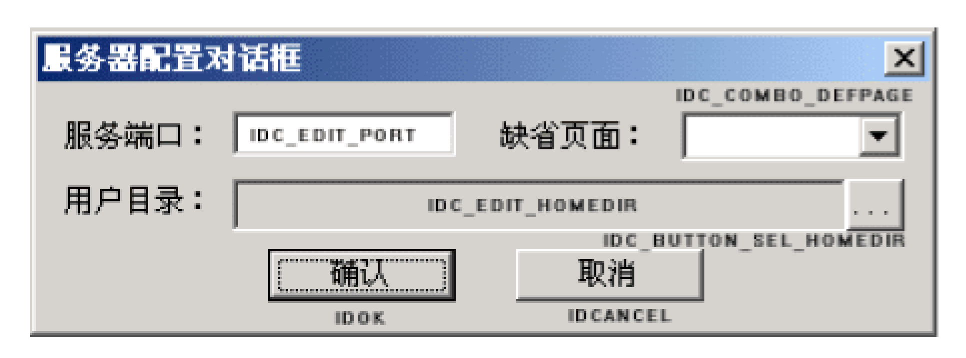


图 22.6　服务器配置对话框设计

下面是 COptSetupDlg 类的头文件：

```
/////////////////////////////////////////////////////////////////////////////
// OptSetupDlg.h : 配置数据操作对话框类的头文件
// Create: 2004-03-25        LastModify: 2004-03-25
/////////////////////////////////////////////////////////////////////////////
#if !defined(AFX_OPTSETUPDLG_H__0248A960_2C27_49BD_8B26_6D9E00727684__INCL
    UDED_)
#define AFX_OPTSETUPDLG_H__0248A960_2C27_49BD_8B26_6D9E00727684__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class COptSetupDlg : public CDialog
{
```

```
// Construction
public:
    COptSetupDlg(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(COptSetupDlg)
    enum { IDD = IDD_DIALOG_OPTSETUP };
    CString     m_strDefaultPage;
    CString     m_strHomeDir;
    UINT         m_nSvrPort;
    //}}AFX_DATA

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(COptSetupDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(COptSetupDlg)
    virtual void OnOK();
    afx_msg void OnButtonSelHomepage();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before
  the previous line.

#endif
/////////////////////////////////////////////////////////////////////////////
```

上面代码中的黑体部分值得读者注意，它们声明了 3 个成员变量和两个成员函数。总的来说，COptSetupDlg 类还是非常简单的，下面直接给出其实现代码：

```
/////////////////////////////////////////////////////////////////////////////
// OptSetupDlg.cpp: 配置数据操作对话框的实现，操作 COptions 类的一个实例
// Create: 2004-03-25        LastModify: 2004-03-25
/////////////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "MyWebServer.h"
#include "OptSetupDlg.h"
#include "Options.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// COptSetupDlg dialog

COptSetupDlg::COptSetupDlg(CWnd* pParent /*=NULL*/)
    : CDialog(COptSetupDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(COptSetupDlg)
    m_strDefaultPage = _T(g_pSvrOptions->m_szDefaultPage);
    m_strHomeDir = _T(g_pSvrOptions->m_szHomeDir);
    m_nSvrPort = g_pSvrOptions->m_nServerPort;
    //}}AFX_DATA_INIT
}

void COptSetupDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(COptSetupDlg)
    DDX_CBString(pDX, IDC_COMBO_DEFPAGE, m_strDefaultPage);
    DDX_Text(pDX, IDC_EDIT_HOMEDIR, m_strHomeDir);
    DDX_Text(pDX, IDC_EDIT_PORT, m_nSvrPort);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(COptSetupDlg, CDialog)
    //{{AFX_MSG_MAP(COptSetupDlg)
    ON_BN_CLICKED(IDC_BUTTON_SEL_HOMEDIR, OnButtonSelHomepage)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// COptSetupDlg message handlers

void COptSetupDlg::OnOK()
{
    // TODO: Add extra validation here
    UpdateData();
    strncpy(g_pSvrOptions->m_szHomeDir, m_strHomeDir, 255);
    strncpy(g_pSvrOptions->m_szDefaultPage, m_strDefaultPage, 255);
    g_pSvrOptions->m_nServerPort = m_nSvrPort;

    CDialog::OnOK();
}
```

```
47  void COptSetupDlg::OnButtonSelHomepage()
48  {
49      // TODO: Add your control notification handler code here
50      BROWSEINFO    bi;
51      char szFolderName[MAX_PATH];
52      char szDirName[MAX_PATH];
53      LPMALLOC lpMalloc;
54      bi.hwndOwner = GetSafeHwnd();
55      bi.pidlRoot = NULL;
56      bi.pszDisplayName = szFolderName;// only folder name
57      bi.lpszTitle = "请选择MyWebWever 的根目录: ";
58      bi.ulFlags = BIF_STATUSTEXT;
59      bi.lpfn = NULL;
60      bi.lParam = NULL;
61      bi.iImage = NULL;
62      LPITEMIDLIST pidl = SHBrowseForFolder(&bi);
63      if(pidl){
64          SHGetPathFromIDList(pidl, szDirName);
65          m_strHomeDir = szDirName;
66          UpdateData(FALSE);
67      }
68
69      if(!SHGetMalloc(&lpMalloc) && (lpMalloc != NULL))
70      {
71          if(pidl != NULL){
72              lpMalloc->Free(pidl);
73          }
74          lpMalloc->Release();
75      }
76  }
////////////////////////////////////////////////////////////////////////////
```

在上面的程序中，黑体字部分是用户代码，其余都由 VC 系统自动生成。下面仅对用户代码进行分析。

第 16～18 行 COptSetupDlg 类构造函数。为成员变量 m_strDefaultPage、m_strHomeDir 和 m_nSvrPort 赋初值，从全局的 COptions 类对象 g_pSvrOptions 获得服务器参数。

第 40～43 行“确认”按钮的响应函数。首先调用 UpdateData 函数将对话框数据读取至对应的成员变量，然后将这些成员变量的值赋给 g_pSvrOptions。

第 50～75 行 IDC_BUTTON_SEL_HOMEDIR 按钮的响应函数。该函数调用系统通用的目录查找对话框供用户选择 MyWeb 的根目录，并将值赋给 strHomeDir 成员变量。

22.2.3　CMyNotifyIcon 类

该类用于构建、维护任务栏图标，主要目的是为用户提供基于任务栏图标的图形化使用界面。

```
//////////////////////////////////////////////////////////////////////////////
// MyNotifyIcon.h: CMyNotifyIcon 类的接口说明
// Create: 2004-04-06      LastModify: 2004-04-06
//////////////////////////////////////////////////////////////////////////////
#ifndef(AFX_MYNOTIFYICON_H__501495DD_2ECE_4241_88A2_3A215EC596A1__INCLUDED_)
#define AFX_MYNOTIFYICON_H__501495DD_2ECE_4241_88A2_3A215EC596A1__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CMyNotifyIcon
{
public:
     void ChangeIcon(UINT nIDResource, const char *tip);
     void DelIcon();
     void AddIcon(UINT nIDResource, const char *tip);
     CMyNotifyIcon(HWND hWnd, UINT uCallbackMessage, UINT nID);
     virtual ~CMyNotifyIcon();

protected:
     UINT m_nID;
     void FillNotifyIconData(UINT nIDResource, const char *tip);
     NOTIFYICONDATA m_niData;
     UINT m_uCallbackMessage;
     HWND m_hWnd;
};
#endif
//////////////////////////////////////////////////////////////////////////////
```

上面是 CMyNotifyIcon 类的接口说明，和用户相关的主要是 CMyNotifyIcon 的构造函数以及 AddIcon、ChangeIcon 与 DelIcon 3 个 public 成员函数，这 3 个函数分别用于添加任务栏图标、切换任务栏图标和删除任务栏图标，所有的操作都围绕函数系统函数 Shell_NotifyIcon 进行。

```
//////////////////////////////////////////////////////////////////////////////
// MyNotifyIcon.cpp: CMyNotifyIcon 类的定义，实现了基本的任务栏图标操作
// Create: 2004-04-06 LastModify: 2004-04-06
//////////////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "MyNotifyIcon.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif
```

```
8    //////////////////////////////////////////////////////////////////////
9    // Construction/Destruction
10   //////////////////////////////////////////////////////////////////////
11   CMyNotifyIcon::CMyNotifyIcon(HWND hWnd, UINT uCallbackMessage, UINT nID)
12   {
13       m_hWnd = hWnd;
14       ASSERT(m_hWnd != NULL);
15       m_nID = nID;
16       m_uCallbackMessage = uCallbackMessage;
17   }

18   CMyNotifyIcon::~CMyNotifyIcon()
19   {
20   }

21   void CMyNotifyIcon::AddIcon(UINT nIDResource, const char *tip)
22   {
23       FillNotifyIconData(nIDResource, tip);
24       Shell_NotifyIcon(NIM_ADD, &m_niData);
25   }

26   void CMyNotifyIcon::ChangeIcon(UINT nIDResource, const char *tip)
27   {
28       FillNotifyIconData(nIDResource, tip);
29       Shell_NotifyIcon(NIM_MODIFY, &m_niData);
30   }

31   void CMyNotifyIcon::DelIcon()
32   {
33       Shell_NotifyIcon(NIM_DELETE, &m_niData);
34   }

35   void CMyNotifyIcon::FillNotifyIconData(UINT nIDResource, const char *tip)
36   {
37       HICON hIcon = AfxGetApp()->LoadIcon(nIDResource);
38       memset(&m_niData, 0, sizeof(NOTIFYICONDATA));
39       m_niData.cbSize = sizeof(NOTIFYICONDATA);
40       m_niData.hIcon = hIcon;
41       m_niData.hWnd = m_hWnd;
42       sprintf(m_niData.szTip, tip);
43       m_niData.uCallbackMessage = m_uCallbackMessage;
44       m_niData.uFlags = NIF_ICON | NIF_MESSAGE | NIF_TIP;
45       m_niData.uID = m_nID;
46   }
/////////////////////////////////////////////////////////////////////////////
```

第 11～17 行 CMyNotifyIcon 类的构造函数，在构造函数中传入 3 个参数：hWnd、

uCallbackMessage 和 nID，分别是使用任务栏图标的应用程序的窗口句柄、任务栏图标鼠标事件的回调用户消息（当有鼠标事件发生时，系统会自动将这个用户定义消息 uCallbackMessage 发送给窗体 hWnd）和任务栏图标的 ID。

第 21～25 行 AddIcon，public 成员函数，添加任务栏图标。函数有两个输入参数：nIDResource 和 tip，分别是图标资源的 ID 号和鼠标停留在任务栏图标上时显示的提示信息。

- 23 行，函数调用 protected 成员函数 FillNotifyIconData 填充 NOTIFYICONDATA 类型成员变量 m_niData。
- 24 行，调用系统函数 Shell_NotifyIcon，指令参数为 NIM_ADD。

第 26～30 行 ChangeIcon，public 成员函数，更改当前的任务栏图标。参数与流程都与 AddIcon 一致，只是在调用系统函数 Shell_NotifyIcon 时指令参数为 NIM_MODIFY。需要注意的是用户必须首先 AddIcon，然后才能 ChangeIcon。

第 31～34 行 DelIcon，public 成员函数，删除任务栏图标，同上，Shell_NotifyIcon 指令参数为 NIM_DELETE。

第 35～46 行 FillNotifyIconData，protected 成员函数，填充 NOTIFYICONDATA 类型成员变量 m_niData。

22.2.4　CHttpServer 类

CHttpServer 类是 MyWeb 项目的核心模块，由它提供全部的 WWW 核心服务，包括接受客户端连接、解析 HTTP 请求信息、返回应答等。代码涉及到完成端口、线程同步、C++标准模板库（STL）等的使用，希望读者在阅读这部分代码之前，对这些知识点有一定的了解。

首先分析 CHttpServer 类的头文件 HttpServer.h，源码如下：

```
////////////////////////////////////////////////////////////////////////////////
// HttpServer.h: CHttpServer 类的接口说明
// Create:        2004-03-23                    By: yangming
// LastModify:    2004-04-05                    By: yangming
////////////////////////////////////////////////////////////////////////////////
#if !defined(AFX_HTTPSERVER_H__1A372664_7D4F_430C_AD6D_2B26CC4CB967__INCLUDED_)
#define AFX_HTTPSERVER_H__1A372664_7D4F_430C_AD6D_2B26CC4CB967__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// The debugger can't handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
#pragma warning(disable:4786)

#include <WINSOCK2.H>
#include <STRING>
```

```
#include <MAP>

using namespace std;

#define MAX_SERVICETHREAD_NUM 8
#define WAIT4THREAD_MILLISECS       3000
#define MAX_BUF_LEN                 1000
#define MAX_HTTP_REQUEST_LEN        1000

#define WEBSERVER_NAME          "MyWeb"
#define ERROR404                string("\\E404.html")
#define ERROR501                string("\\E501.html")

#define WM_USER_CLIENT          WM_USER + 1
#define CLIENT_CONN_REQ         0
#define CLIENT_ACCEPT           1
#define CLIENT_REJECT           2
#define CLIENT_DISCONNECT       3
#define CLIENT_FAIL_CLOSE       4
typedef enum{
    SERVER_STOP, SERVER_RUNNING, SERVER_PAUSE
} ServerState;

typedef map<string, string>            MIMETYPES;

/**********************************************************************/
typedef enum _IO_OPER{
   SVR_IO_READ,
   SVR_IO_WRITE
} IO_OPER, *LPIO_OPER;

// 扩展重叠结构体，单 I/O 数据
typedef struct _OverLappedEx{
    OVERLAPPED              OverLapped;
    WSABUF                  wbuf;
    char                data[MAX_BUF_LEN];
    IO_OPER                 oper;
    DWORD                   flags;
} PER_IO_DATA, *LPPER_IO_DATA;

// 完成键结构体，单句柄数据，对应每个服务套接口—每个连接
typedef struct _CONN_CTX{
    SOCKET                  sockAccept;
    LPPER_IO_DATA           pPerIOData;
    string                  szRequest;
    string                  szResponse;
    BOOL                bKeepAlive;
    DWORD                   nAlreadyResponsed;
```

```
    struct _CONN_CTX      *pPrec;
    struct _CONN_CTX      *pNext;
    _CONN_CTX(){ pPerIOData = (LPPER_IO_DATA) malloc(sizeof(PER_IO_DATA));};
    ~_CONN_CTX(){ free(pPerIOData);};
} CONN_CTX, *LPCONN_CTX;
/*********************************************************************/

class CHttpServer
{
public:
    BOOL StartServer();
    BOOL PauseServer();
    BOOL StopServer();
    ServerState GetServerState();
    CHttpServer();
    virtual ~CHttpServer();

protected:
    HANDLE m_evtSvrToStop;
    HANDLE m_evtThreadLanched;
    HANDLE m_hIOCP;
    HANDLE m_hThreadList[MAX_SERVICETHREAD_NUM + 2];
        // MAX_SERVICETHREAD_NUM Service Thread, one ListenThread & one AdminThread
    ServerState m_ServerState;
    SOCKET m_sdListen;
    LPCONN_CTX   m_ptrConnCtxHead; // 双向链表，用于保存服务器所有连接信息
    BOOL m_bSvrPaused;
    CRITICAL_SECTION m_CriticalSection;
    MIMETYPES m_MimeTypes;
    UINT m_nSvcThreadNum;

protected:
    void InitMimeTypes();
    BOOL ProcessRequest(string szRequest, string &szResponse, BOOL &bKeepAlive);
    BOOL IsRequestCompleted(string szRequest);
    void ResetAll();
    LPCONN_CTX CreateConnCtx(SOCKET sockAccept, HANDLE hIOCP); // 创建连接数据信息
    void ConnListAdd(LPCONN_CTX lpConnCtx);
    void ConnListRemove(LPCONN_CTX lpConnCtx);
    void ConnListClear();

    static DWORD WINAPI ListenThread(LPVOID pParam);
    static DWORD WINAPI ServiceThread(LPVOID pParam);
    static DWORD WINAPI AdminThread(LPVOID pParam);
};
```




```
93   extern CHttpServer *g_pHttpServer;

94   #endif
//////////////////////////////////////////////////////////////////////////
```

第 9 行在本节开始部分，我们说过 CHttpServer 应用了 C++的标准模板库。由于 STL 创建了许多长符号（symbols），而 VC 调试器仅能处理不多于 255 个字符的符号，因此在编译应用了 STL 的项目时，会产生大量的编码为 4786 的警告信息。解决的方法如下：在 include 所需的 STL 库之前，首先通过 pragma 设定编译选项以禁止 4786 警告。

第 11～13 行包含 STL 中 string 和 map 头文件，并设定 std 名字空间。

第 14～20 行定义了若干常量。其中 MAX_SERVICETHREAD_NUM（=8）表示服务器最多只启动 8 个服务线程；WAIT4THREAD_MILLISECS（=3000）用于线程间的同步，表示线程 A 在等待线程 B 触发某事件对象时，最多只等待 3 秒钟（或若干个 3 秒钟，详见 CHttpServer 的实现）；MAX_BUF_LEN（=1000），表示 MyWeb 服务器每次最多接收客户端传来的 1000B 数据；MAX_HTTP_REQUEST_LEN（=1000），表示 MyWeb 接受的客户端 HTTP 请求最大长度为 1000B；WEBSERVER_NAME（="MyWeb"），用于构造 HTTP 的 Response 信息，表示服务器名为 MyWeb；ERROR404（=string("\\E404.html")）和 ERROR501（=string("\\E501.html")）用于指定当客户端请求发生 404 或者 501 错误时，MyWeb 服务器所反馈的页面。

第 21～26 行同样定义了若干常量。我们知道，每一个 Windows 消息都带有三部分数据：消息码、WPARAM 和 LPARAM。当 MyWeb 服务过程中发生了某些与 HTTP 客户端相关的事件时，CMyHttpServer 模块会自动向主窗体发送 WM_USER_CLIENT 消息，并且将 CLIENT_CONN_REQ、CLIENT_ACCEPT、CLIENT_REJECT、CLIENT_DISCONNECT 以及 CLIENT_FAIL_CLOSE 作为 LPARAM 发送（客户端 IP 信息作为 WPARAM）；这些 CLIENT_XXX 常量分别表示有客户端发起连接请求、接受连接、拒绝连接、客户端关闭连接以及有错误发生服务器切断连接等。

第 27～29 行定义了枚举类型 ServerState，用于表示服务器的 3 个基本状态：SERVER_STOP、SERVER_RUNNING 和 SERVER_PAUSE。

第 30 行将 map<string, string>定义为 MIMETYPES 类型。

第 32～35 行定义了枚举类型 IO_OPER，对应于读/写两种操作。

第 37～43 行定义了完成端口模型中的单 I/O 操作数据结构，该结构与程序 21.3 中的定义完全一致，简述如下：OverLapped，重叠结构体；wbuf，用于本次 I/O 操作的 WSABUF 结构；data，实际的用户 I/O 缓冲区，在 MyWeb 系统中仅用于接收操作；oper，用于标志 I/O 操作的类型；flags，用于设置 I/O 操作的参数。在 CHttpServer 模块中，PER_IO_DATA 依然作为普通的结构体进行操作，这与下面的完成键有所不同。

第 45～56 行完成键结构。该结构与程序 21.3 中的定义有较大差别，最明显的是此处的结构体包含了构造函数和析构函数；此外，在程序 21.3 中，完成键结构体的内存处理工作由 malloc/free 函数完成，而在 CHttpServer 模块中由 new/delete 函数完成。事实上，在前面的定义中 LPCONN_CTX 是指向一块内存区的指针，而在此处是类 CONN_CTX 的对象指针。

- 46 行，sockAccept，用于保存此连接对应的服务套接口描述字。
- 47 行，pPerIOData，指向单 I/O 操作结构的指针，用于保存每次 I/O 操作的基本信息和数据。该指针所指向的内存区的分配和释放工作由_CONN_CTX 的构造/析构函数完成。
- 48～49 行，两个 string 类型的成员变量，其中 szRequest 用于保存 HTTP 请求信息，szResponse 用于保存服务器对请求的响应数据。string 类由 STL 提供，包含 erase、size 等成员函数。正是因为在 CONN_CTX 中包含了 string 类型的成员变量，不能使用 malloc/free 函数来为 LPCONN_CTX 指针分配内存（否则将导致内存溢出，szRequest 和 szResponse 无法正常释放资源），CHttpServer 模块才将 CONN_CTX 当作类来处理。
- 50 行，bKeepAlive 用于指明该连接是否需要保持，参见相关的 RFC 文档。
- 51 行，由于 HTTP 的响应信息数据量较大，可能需要多次 WSASend 调用才能发送完毕，成员变量 nAlreadyResponsed 用于记录当前连接对当前请求已发送的响应数据量。
- 52～53 行，_CONN_CTX 类指针，用于构造包含当前所有连接的 CONN_CTX 双向链表。
- 54～55 行，_CONN_CTX 的构造/析构函数，在构造函数中，调用了 malloc 函数为 pPerIOData 分配内存；在析构函数中，调用 free 释放内存。因此 pPerIOData 的内存处理是在 CONN_CTX 对象创建和释放时自动进行的。这一点与程序 21.3 不同，在 21.3 中，必须显式地按一定顺序分别为 LPPER_IO_DATA 和 LPCONN_CTX 类型指针调用 malloc/free 函数。

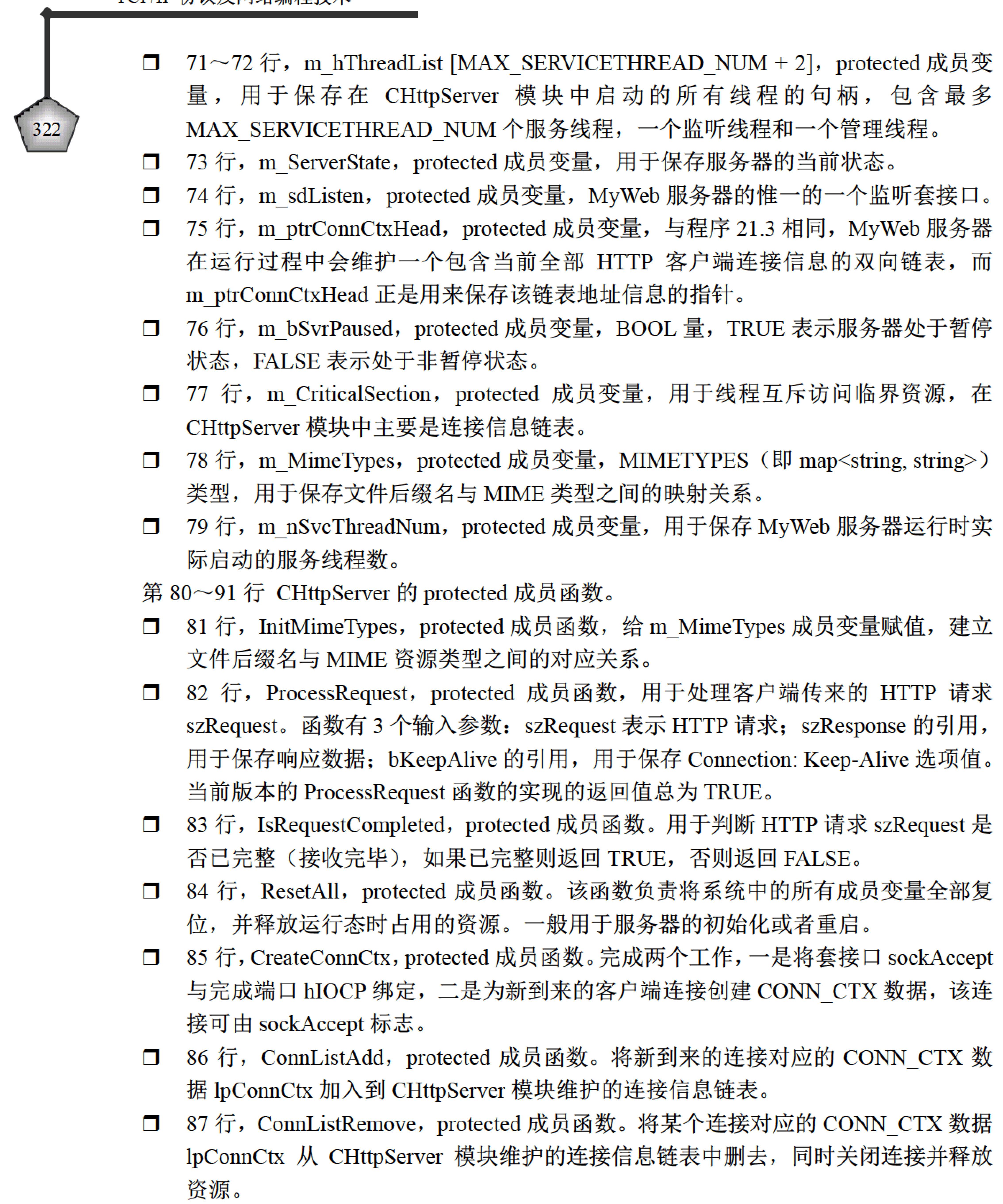
第 58～92 行 CHttpServer 类定义。主要分为三段，60～66 行，是类的 public 成员函数，用户使用接口；67～79 行，类的 protected 成员变量；80～91 行，类的 protected 成员函数，供类内部调用。

第 60～66 行 CHttpServer 的 public 成员函数。

- 61 行，StartServer，public 成员函数，用于启动 MyWeb 服务。如果函数调用返回 TRUE，说明服务成功启动；如果返回 FALSE，则需要根据服务器的当前状态来判断，有两种情况：① 服务器原本就处于运行态；② 服务器启动失败。
- 62 行，PauseServer，public 成员函数，用于暂停服务。如果函数调用返回 TRUE，说明服务已暂停；如果返回 FALSE，说明服务器原本就处于暂停或者停止状态。
- 63 行，StopServer，public 成员函数，用于停止服务。如果函数调用返回 TRUE，说明服务成功停止；如果返回 FALSE，说明服务器原本就处于停止状态。
- 64 行，GetServerState 用于返回服务器的当前状态，返回类型为 ServerState，枚举型，见第 27～29 行。

第 67～79 行 CHttpServer 的 protected 成员变量。

- 68～69 行，m_evtSvrToStop 和 m_evtThreadLanched，protected 成员变量。事件对象，用于线程同步。其中 evtSvrToStop 用于通知管理线程（见下文）停止 MyWeb 服务；m_evtThreadLanched 用于通知线程已成功启动。
- 70 行，m_hIOCP，完成端口句柄。



- 71～72 行，m_hThreadList [MAX_SERVICETHREAD_NUM + 2]，protected 成员变量，用于保存在 CHttpServer 模块中启动的所有线程的句柄，包含最多 MAX_SERVICETHREAD_NUM 个服务线程，一个监听线程和一个管理线程。
- 73 行，m_ServerState，protected 成员变量，用于保存服务器的当前状态。
- 74 行，m_sdListen，protected 成员变量，MyWeb 服务器的惟一的一个监听套接口。
- 75 行，m_ptrConnCtxHead，protected 成员变量，与程序 21.3 相同，MyWeb 服务器在运行过程中会维护一个包含当前全部 HTTP 客户端连接信息的双向链表，而 m_ptrConnCtxHead 正是用来保存该链表地址信息的指针。
- 76 行，m_bSvrPaused，protected 成员变量，BOOL 量，TRUE 表示服务器处于暂停状态，FALSE 表示处于非暂停状态。
- 77 行，m_CriticalSection，protected 成员变量，用于线程互斥访问临界资源，在 CHttpServer 模块中主要是连接信息链表。
- 78 行，m_MimeTypes，protected 成员变量，MIMETYPES（即 map<string, string>）类型，用于保存文件后缀名与 MIME 类型之间的映射关系。
- 79 行，m_nSvcThreadNum，protected 成员变量，用于保存 MyWeb 服务器运行时实际启动的服务线程数。

第 80～91 行 CHttpServer 的 protected 成员函数。

- 81 行，InitMimeTypes，protected 成员函数，给 m_MimeTypes 成员变量赋值，建立文件后缀名与 MIME 资源类型之间的对应关系。
- 82 行，ProcessRequest，protected 成员函数，用于处理客户端传来的 HTTP 请求 szRequest。函数有 3 个输入参数：szRequest 表示 HTTP 请求；szResponse 的引用，用于保存响应数据；bKeepAlive 的引用，用于保存 Connection: Keep-Alive 选项值。当前版本的 ProcessRequest 函数的实现的返回值总为 TRUE。
- 83 行，IsRequestCompleted，protected 成员函数。用于判断 HTTP 请求 szRequest 是否已完整（接收完毕），如果已完整则返回 TRUE，否则返回 FALSE。
- 84 行，ResetAll，protected 成员函数。该函数负责将系统中的所有成员变量全部复位，并释放运行态时占用的资源。一般用于服务器的初始化或者重启。
- 85 行，CreateConnCtx，protected 成员函数。完成两个工作，一是将套接口 sockAccept 与完成端口 hIOCP 绑定，二是为新到来的客户端连接创建 CONN_CTX 数据，该连接可由 sockAccept 标志。
- 86 行，ConnListAdd，protected 成员函数。将新到来的连接对应的 CONN_CTX 数据 lpConnCtx 加入到 CHttpServer 模块维护的连接信息链表。
- 87 行，ConnListRemove，protected 成员函数。将某个连接对应的 CONN_CTX 数据 lpConnCtx 从 CHttpServer 模块维护的连接信息链表中删去，同时关闭连接并释放资源。
- 88 行，ConnListClear，protected 成员函数。清除连接信息链表，关闭所有连接。
- 89～91 行，ListenThread/ServiceThread/AdminThread，这 3 个函数既是 CHttpServer 类的 protected 成员函数，又是线程函数。需要注意的是，类的成员函数要作为线程函数，必须是静态的并且带有 WINAPI 描述符。而又因为 ListenThread/ServiceThread

/AdminThread 是静态函数，所以在这 3 个函数中无法使用 this 指针，也就无法访问类的非静态成员变量或者调用非静态成员函数。解决的方法是将 this 指针作为线程函数的输入参数 pParam 传入。其中 ListenThread 是 CHttpServer 模块的监听线程，负责接收外来的连接请求；ServiceThread 是服务线程，由它具体接收并处理客户端发送的 HTTP 请求；AdminThread 是管理线程，该线程接收服务器停止信号，并负责服务器的停止、复位工作。

下面是 CHttpServer 类的实现源码：

```
//////////////////////////////////////////////////////////////////////
// HttpServer.cpp: CHttpServer 类的实现，完成了基于 IOCP 的简单的 WWW 服务功能
// Create:         2004-03-23                    By: yangming
// LastModify:     2004-04-05                    By: yangming
//////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "MyWebServer.h"
#include "HttpServer.h"
#include "Options.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

CHttpServer     _HttpServer;
CHttpServer     *g_pHttpServer = &_HttpServer;

// 创建 I/O 完成端口
#define CreateNewIoCompletionPort(dwNumberOfConcurrentThreads)
   CreateIoCompletionPort (INVALID_HANDLE_VALUE, NULL, 0,
   dwNumberOfConcurrentThreads)
// 将套接口与完成端口绑定
#define AssociateWithIoCompletionPort(hComPort, hDevice, dwCompKey)
   CreateIoCompletionPort(hDevice, hComPort, dwCompKey, 0)

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CHttpServer::CHttpServer()
{
    m_evtThreadLanched = CreateEvent(NULL, FALSE, FALSE, NULL);
    m_evtSvrToStop = CreateEvent(NULL, FALSE, FALSE, NULL);
    InitializeCriticalSection(&m_CriticalSection);
    ResetAll();

    SYSTEM_INFO sysInfo;
```

```
    GetSystemInfo(&sysInfo);
    // 将 sysInfo.dwNumberOfProcessors*2 和 MAX_SERVICETHREAD_NUM 之间的较小值赋
       给 m_nSvcThreadNum
    m_nSvcThreadNum=sysInfo.dwNumberOfProcessors*2<MAX_SERVICETHREAD_NUM?
       (sysInfo.dwNumberOfProcessors * 2) : MAX_SERVICETHREAD_NUM;

    InitMimeTypes();
}

CHttpServer::~CHttpServer()
{
    CloseHandle(m_evtThreadLanched);
    CloseHandle(m_evtSvrToStop);
    DeleteCriticalSection(&m_CriticalSection);
}

BOOL CHttpServer::StartServer()
{
    if(m_ServerState == SERVER_RUNNING)
        return FALSE;
    if(m_ServerState == SERVER_PAUSE){
        m_bSvrPaused = FALSE;
        m_ServerState = SERVER_RUNNING;
        return TRUE;
    }
    m_bSvrPaused = FALSE;

    // Step.1 初始化 Winsock
    WSAData wsaData;
    if(WSAStartup(WINSOCK_VERSION, &wsaData) != 0)
        return FALSE;
    ResetAll();

    // Step.2 启动管理线程，该线程负责服务器的关闭工作
    HANDLE hThread = CreateThread(NULL, 0, AdminThread, this, 0, NULL);
    if(hThread == NULL){
        WSACleanup();
        ResetAll();
        return FALSE;
    }
    // 等待管理线程正常运行
    if(WaitForSingleObject(m_evtThreadLanched, WAIT4THREAD_MILLISECS) !=
       WAIT_OBJECT_0){
        TerminateThread(hThread, 1);
        CloseHandle(hThread);
        WSACleanup();
```

```
        ResetAll();
        return FALSE;
    }

    m_hThreadList[0] = hThread;


    // Step.3 启动监听线程
    ResetEvent(m_evtThreadLanched);
    hThread = CreateThread(NULL, 0, ListenThread, this, 0, NULL);
    if(hThread == NULL){
        // 通知管理线程结束
        SetEvent(m_evtSvrToStop);
        // 等待管理线程结束
        if(WaitForSingleObject(m_hThreadList[0], WAIT4THREAD_MILLISECS * 3) !
           = WAIT_OBJECT_0)
            TerminateThread(m_hThreadList[0], 1);
        CloseHandle(m_hThreadList[0]);
        WSACleanup();
        ResetAll();
        return FALSE;
    }
    // 等待监听线程正常运行
    if(WaitForSingleObject(m_evtThreadLanched, WAIT4THREAD_MILLISECS) !=
      WAIT_OBJECT_0){
        TerminateThread(hThread, 1);
        CloseHandle(hThread);
        SetEvent(m_evtSvrToStop);// 通知管理线程结束
        if(WaitForSingleObject(m_hThreadList[0], WAIT4THREAD_MILLISECS *
           3) != WAIT_OBJECT_0)
            TerminateThread(m_hThreadList[0], 1);
        CloseHandle(m_hThreadList[0]);
        WSACleanup();
        ResetAll();
        return FALSE;
    }

    m_hThreadList[1] = hThread;
    m_ServerState = SERVER_RUNNING;

    return TRUE;
}

BOOL CHttpServer::PauseServer()
{
    if(m_ServerState != SERVER_RUNNING)
        return FALSE;
```

```
    m_bSvrPaused = TRUE;
    m_ServerState = SERVER_PAUSE;

    return TRUE;
}

BOOL CHttpServer::StopServer()
{
    if(m_ServerState == SERVER_STOP)
        return FALSE;

    SetEvent(m_evtSvrToStop);
    // 等待管理线程结束
    WaitForSingleObject(m_hThreadList[0], WAIT4THREAD_MILLISECS * 3);
    DWORD nExitCode;
    GetExitCodeThread(m_hThreadList[0], &nExitCode);
    if(nExitCode == STILL_ACTIVE)
        TerminateThread(m_hThreadList[0], 1);

    ResetAll();
    WSACleanup();

    return TRUE;
}

ServerState CHttpServer::GetServerState()
{
    return m_ServerState;
}


/////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////

void CHttpServer::InitMimeTypes()
{
    // Init MIME Types
    m_MimeTypes["doc"]            = "application/msword";
    m_MimeTypes["bin"]            = "application/octet-stream";
    m_MimeTypes["dll"]            = "application/octet-stream";
    m_MimeTypes["exe"]            = "application/octet-stream";
    m_MimeTypes["pdf"]            = "application/pdf";
    m_MimeTypes["p7c"]            = "application/pkcs7-mime";
    m_MimeTypes["ai"]             = "application/postscript";
    m_MimeTypes["eps"]            = "application/postscript";
    m_MimeTypes["ps"]             = "application/postscript";
    m_MimeTypes["rtf"]            = "application/rtf";
    m_MimeTypes["fdf"]            = "application/vnd.fdf";
```

```
m_MimeTypes["arj"]          = "application/x-arj";
m_MimeTypes["gz"]           = "application/x-gzip";
m_MimeTypes["class"]        = "application/x-java-class";
m_MimeTypes["js"]           = "application/x-javascript";
m_MimeTypes["lzh"]          = "application/x-lzh";
m_MimeTypes["lnk"]          = "application/x-ms-shortcut";
m_MimeTypes["tar"]          = "application/x-tar";
m_MimeTypes["hlp"]          = "application/x-winhelp";
m_MimeTypes["cert"]         = "application/x-x509-ca-cert";
m_MimeTypes["zip"]          = "application/zip";
m_MimeTypes["cab"]          = "application/x-compressed";
m_MimeTypes["arj"]          = "application/x-compressed";
m_MimeTypes["aif"]          = "audio/aiff";
m_MimeTypes["aifc"]         = "audio/aiff";
m_MimeTypes["aiff"]         = "audio/aiff";
m_MimeTypes["au"]           = "audio/basic";
m_MimeTypes["snd"]          = "audio/basic";
m_MimeTypes["mid"]          = "audio/midi";
m_MimeTypes["rmi"]          = "audio/midi";
m_MimeTypes["mp3"]          = "audio/mpeg";
m_MimeTypes["vox"]          = "audio/voxware";
m_MimeTypes["wav"]          = "audio/wav";
m_MimeTypes["ra"]           = "audio/x-pn-realaudio";
m_MimeTypes["ram"]          = "audio/x-pn-realaudio";
m_MimeTypes["bmp"]          = "image/bmp";
m_MimeTypes["gif"]          = "image/gif";
m_MimeTypes["jpeg"]         = "image/jpeg";
m_MimeTypes["jpg"]          = "image/jpeg";
m_MimeTypes["tif"]          = "image/tiff";
m_MimeTypes["tiff"]         = "image/tiff";
m_MimeTypes["xbm"]          = "image/xbm";
m_MimeTypes["wrl"]          = "model/vrml";
m_MimeTypes["htm"]          = "text/html";
m_MimeTypes["html"]         = "text/html";
m_MimeTypes["c"]            = "text/plain";
m_MimeTypes["cpp"]          = "text/plain";
m_MimeTypes["def"]          = "text/plain";
m_MimeTypes["h"]            = "text/plain";
m_MimeTypes["txt"]          = "text/plain";
m_MimeTypes["rtx"]          = "text/richtext";
m_MimeTypes["rtf"]          = "text/richtext";
m_MimeTypes["java"]         = "text/x-java-source";
m_MimeTypes["css"]          = "text/css";
m_MimeTypes["mpeg"]         = "video/mpeg";
m_MimeTypes["mpg"]          = "video/mpeg";
m_MimeTypes["mpe"]          = "video/mpeg";
m_MimeTypes["avi"]          = "video/msvideo";
m_MimeTypes["mov"]          = "video/quicktime";
```

```
    m_MimeTypes["qt"]              = "video/quicktime";
    m_MimeTypes["shtml"]           = "wwwserver/html-ssi";
    m_MimeTypes["asa"]             = "wwwserver/isapi";
    m_MimeTypes["asp"]             = "wwwserver/isapi";
    m_MimeTypes["cfm"]             = "wwwserver/isapi";
    m_MimeTypes["dbm"]             = "wwwserver/isapi";
    m_MimeTypes["isa"]             = "wwwserver/isapi";
    m_MimeTypes["plx"]             = "wwwserver/isapi";
    m_MimeTypes["url"]             = "wwwserver/isapi";
    m_MimeTypes["cgi"]             = "wwwserver/isapi";
    m_MimeTypes["php"]             = "wwwserver/isapi";
    m_MimeTypes["wcgi"]            = "wwwserver/isapi";
}

void CHttpServer::ResetAll()
{
    m_ServerState = SERVER_STOP;
    m_sdListen = INVALID_SOCKET;
    for(int i = 0; i < MAX_SERVICETHREAD_NUM + 2; i++){
        if(m_hThreadList[i])
            CloseHandle(m_hThreadList[i]);
        m_hThreadList[i] = NULL;
    }
    if(m_hIOCP){
        CloseHandle(m_hIOCP);
        m_hIOCP = NULL;
    }
    if(m_ptrConnCtxHead)
        ConnListClear();
    ResetEvent(m_evtThreadLanched);
    ResetEvent(m_evtSvrToStop);
    m_bSvrPaused = FALSE;
    m_ptrConnCtxHead = NULL;
}

////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////

BOOL CHttpServer::IsRequestCompleted(string szRequest)
{
    if(szRequest.size() < 4)
        return FALSE;

    if(szRequest.substr(szRequest.size() - 4, 4) == "\r\n\r\n")
        return TRUE;
    else
        return FALSE;
}
```

```
BOOL CHttpServer::ProcessRequest(string szRequest, string &szResponse, BOOL
   &bKeepAlive)
{
     string szMethod;
     string szFileName;
     string szFullPathName;
     string szFileExt;
     string szStatusCode("200 OK");
     string szContentType("text/html");
     string szConnectionType("close");
     string szNotFoundMessage;
     string szDateTime;
     char pResponseHeader[2048];
     fpos_t lengthActual = 0, length = 0;
     char *pBuf = NULL;
     int n;

     // Check Method
     n = szRequest.find(" ", 0);
     if(n != string::npos){
          szMethod = szRequest.substr(0, n);
          if(szMethod == "GET"){
               // Get file name
               int n1 = szRequest.find(" ", n + 1);
               if(n1 != string::npos){
                    szFileName = szRequest.substr(n + 1, n1 - n - 1);
                    // do some check
                    n1 = szFileName.find("..", 0);
                    if(n1 != string::npos){
                         szStatusCode = "404 Resource not found";
                         szFileName = ERROR404;
                    }
                    if(szFileName == "/")
                         szFileName = string(g_pSvrOptions->m_szDefaultPage);
               }
               else{// No 'space' found in Request String
                    szStatusCode = "501 Not Implemented";
                    szFileName = ERROR501;
               }
          }
          else{
               szStatusCode = "501 Not Implemented";
               szFileName = ERROR501;
          }
     }
     else{// "No 'space' found in Request String
          szStatusCode = "501 Not Implemented";
```

```
        szFileName = ERROR501;
    }

    // Determine Connection type
    n = szRequest.find("\nConnection: Keep-Alive", 0);
    if(n != string::npos)
        bKeepAlive = TRUE;
    else
        bKeepAlive = FALSE;

    // Figure out content type
    int nPointPos = szFileName.rfind(".");
    if(nPointPos != string::npos){
        szFileExt = szFileName.substr(nPointPos + 1, szFileName.size());
        strlwr((char*)szFileExt.c_str());
        MIMETYPES::iterator it;
        it = m_MimeTypes.find(szFileExt);
        if(it != m_MimeTypes.end())
            szContentType = (*it).second;
    }

    // Obtain current GMT date/time
    char szDT[128];
    struct tm *newtime;
    long ltime;
    time(&ltime);
    newtime = gmtime(&ltime);
    strftime(szDT, 128, "%a, %d %b %Y %H:%M:%S GMT", newtime);


    // Read the file
    FILE *f;
    szFullPathName = string(g_pSvrOptions->m_szHomeDir) + "\\" + szFileName;
    f = fopen(szFullPathName.c_str(), "r+b");
    if(f != NULL){
        // Retrive file size
        fseek(f, 0, SEEK_END);
        fgetpos(f, &lengthActual);
        fseek(f, 0, SEEK_SET);

        pBuf = new char[lengthActual + 1];

        length = fread(pBuf, 1, lengthActual, f);
        fclose(f);

        // Make Response
        sprintf(pResponseHeader, "HTTP/1.0 %s\r\nDate: %s\r\nServer: %s\r\
           nAccept-Ranges: bytes\r\nContent-Length: %d\r\nConnection: %s\r\
```

```
                nContent-Type: %s\r\n\r\n",
                szStatusCode.c_str(), szDT, WEBSERVER_NAME, (int)length,
                bKeepAlive ? "Keep-Alive" : "close", szContentType.c_str());
    }
    else{
        // In case of file not found
        szFullPathName = g_pSvrOptions->m_szHomeDir + ERROR404;
        f = fopen(szFullPathName.c_str(), "r+b");
        if(f != NULL){
            // Retrive file size
            fseek(f, 0, SEEK_END);
            fgetpos(f, &lengthActual);
            fseek(f, 0, SEEK_SET);
            pBuf = new char[lengthActual + 1];
            length = fread(pBuf, 1, lengthActual, f);
            fclose(f);
            szNotFoundMessage = string(pBuf, length);
            delete [] pBuf;
            pBuf = NULL;
        }
        szStatusCode = "404 Resource not found";

        sprintf(pResponseHeader, "HTTP/1.0 %s\r\nContent-Length: %d\r\
           nContent-Type: text/html\r\nDate: %s\r\nServer: %s\r\n\r\n%s",
            szStatusCode.c_str(), szNotFoundMessage.size(), szDT, WEBSERVER
                _NAME, szNotFoundMessage.c_str());
        bKeepAlive = FALSE;
    }

    szResponse = string(pResponseHeader);
    if(pBuf)
        szResponse += string(pBuf, length);
    delete [] pBuf;
    pBuf = NULL;

    return TRUE;
}


/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////

LPCONN_CTX CHttpServer::CreateConnCtx(SOCKET sockAccept, HANDLE hIOCP)
{
    LPCONN_CTX lpConnCtx = new CONN_CTX;
    if(lpConnCtx == NULL)
        return NULL;
```

```
    // 赋值
    lpConnCtx->pNext = NULL;
    lpConnCtx->pPrec = NULL;
    lpConnCtx->sockAccept = sockAccept;
    lpConnCtx->szRequest.erase(0, string::npos);
    lpConnCtx->szResponse.erase(0, string::npos);
    lpConnCtx->bKeepAlive = FALSE;
    lpConnCtx->nAlreadyResponsed = 0;

    ZeroMemory(lpConnCtx->pPerIOData, sizeof(PER_IO_DATA));
    lpConnCtx->pPerIOData->OverLapped.hEvent = NULL;
    lpConnCtx->pPerIOData->OverLapped.Internal = 0;
    lpConnCtx->pPerIOData->OverLapped.InternalHigh = 0;
    lpConnCtx->pPerIOData->OverLapped.Offset = 0;
    lpConnCtx->pPerIOData->OverLapped.OffsetHigh = 0;
    lpConnCtx->pPerIOData->wbuf.buf = (char *) lpConnCtx->pPerIOData->data;
    lpConnCtx->pPerIOData->wbuf.len = MAX_BUF_LEN;
    lpConnCtx->pPerIOData->oper = SVR_IO_READ;
    lpConnCtx->pPerIOData->flags = 0;

    // 将套接口与完成端口绑定
    if(!AssociateWithIoCompletionPort(m_hIOCP, (HANDLE) sockAccept, (DWORD)
       lpConnCtx)){
        delete lpConnCtx;
        lpConnCtx = NULL;
        return NULL;
    }

    return lpConnCtx;
}

void CHttpServer::ConnListAdd(LPCONN_CTX lpConnCtx)
{
    LPCONN_CTX    pTemp;

    EnterCriticalSection(&m_CriticalSection);

    if(m_ptrConnCtxHead == NULL) {
        // 链表的第一个（惟一）节点
        lpConnCtx->pPrec = NULL;
        lpConnCtx->pNext = NULL;
        m_ptrConnCtxHead = lpConnCtx;
    }else{
        // 加到链表头部
        pTemp = m_ptrConnCtxHead;
        m_ptrConnCtxHead = lpConnCtx;
        lpConnCtx->pNext = pTemp;
        lpConnCtx->pPrec = NULL;
```

```
            pTemp->pPrec = lpConnCtx;
        }

        LeaveCriticalSection(&m_CriticalSection);
}

void CHttpServer::ConnListRemove(LPCONN_CTX lpConnCtx)
{
        LPCONN_CTX pPrec;
        LPCONN_CTX pNext;

        EnterCriticalSection(&m_CriticalSection);
        if(lpConnCtx) {
            pPrec = lpConnCtx->pPrec;
            pNext = lpConnCtx->pNext;
            if((pPrec == NULL) && (pNext == NULL)){// [*] -> NULL: 链表惟一节点
                m_ptrConnCtxHead = NULL;
            }
            else if((pPrec == NULL) && (pNext != NULL)){// [*] -> [ ] -> .... [ ]: 链表首节点
                pNext->pPrec = NULL;
                m_ptrConnCtxHead = pNext;
            }
            else if((pPrec != NULL) && (pNext == NULL)){// [ ] -> [ ] -> .... [*]: 链表末节点
                pPrec->pNext = NULL;
            }
            else if( pPrec && pNext ) {// [ ] -> [*] -> .... [ ]: 链表中间节点
                pPrec->pNext = pNext;
                pNext->pPrec = pPrec;
            }

            // 关闭连接，释放资源
            if(lpConnCtx->sockAccept != INVALID_SOCKET){
#ifdef _DEMO
                SOCKADDR_IN from;
                memset(&from, 0, sizeof(from));
                int fromlen = sizeof(from);
                getpeername(lpConnCtx->sockAccept, (SOCKADDR *) &from, &fromlen);
                AfxGetMainWnd()->PostMessage(WM_USER_CLIENT, from.sin_addr.s_addr,
                   CLIENT_DISCONNECT);
#endif
                closesocket(lpConnCtx->sockAccept);
            }
            delete lpConnCtx;
            lpConnCtx = NULL;
        }

```

```
    LeaveCriticalSection(&m_CriticalSection);
    return;
}

void CHttpServer::ConnListClear()
{
    LPCONN_CTX pTemp1, pTemp2;

    EnterCriticalSection(&m_CriticalSection);
    pTemp1 = m_ptrConnCtxHead;
    while(pTemp1) {
        pTemp2 = pTemp1->pNext;
        ConnListRemove(pTemp1);
        pTemp1 = pTemp2;
    }
    LeaveCriticalSection(&m_CriticalSection);
    return;
}

/////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////

// 服务器管理线程
DWORD WINAPI CHttpServer::AdminThread(LPVOID pParam)
{
    CHttpServer *pHttpServer = (CHttpServer *) pParam;
    // 通知 StartServer 函数，Admin 线程已启动
    SetEvent(pHttpServer->m_evtThreadLanched);
    // 等待服务停止指令，该指令在三种情况下会被触发:
    // 1. 调用 StartServer 函数时，ListenThread 未能正常启动
    // 2. 服务器 ListenThread 运行过程中出错
    // 3. 调用 StopServer 函数
    WaitForSingleObject(pHttpServer->m_evtSvrToStop, INFINITE);
    // Do Clear Work
    if(pHttpServer->m_sdListen != INVALID_SOCKET)
        closesocket(pHttpServer->m_sdListen);// 将导致 ListenThread 的 accept
           调用出错并结束
    if(pHttpServer->m_hIOCP){// 通知 Service 线程结束
        for(UINT i = 0; i < pHttpServer->m_nSvcThreadNum; i++)
            PostQueuedCompletionStatus(pHttpServer->m_hIOCP, 0, 0, NULL);
    }
    // 等待所有 Service 线程和监听线程结束
    WaitForMultipleObjects(pHttpServer->m_nSvcThreadNum + 1,
        &(pHttpServer->m_hThreadList[1]),
        TRUE,
        WAIT4THREAD_MILLISECS * 2);

    DWORD nExitCode;
```

```
    for(UINT j = 1; j < pHttpServer->m_nSvcThreadNum + 1; j++){
        GetExitCodeThread(pHttpServer->m_hThreadList[j], &nExitCode);
        if(nExitCode == STILL_ACTIVE)
            TerminateThread(pHttpServer->m_hThreadList[j], 1);
    }

    pHttpServer->ResetAll();
    return 0;
}

// 服务器监听线程
DWORD WINAPI CHttpServer::ListenThread(LPVOID pParam)
{
    CHttpServer *pHttpServer = (CHttpServer *) pParam;

    // 创建 I/O 完成端口
    pHttpServer->m_hIOCP = CreateNewIoCompletionPort(0);
    if(pHttpServer->m_hIOCP == NULL)
        return -1;


    // 创建多个工作线程
    for(UINT i = 0; i < pHttpServer->m_nSvcThreadNum; i++){
        HANDLE hThread = CreateThread(NULL, 0, ServiceThread, pHttpServer,
           0, NULL);
        if(hThread == NULL)
            return -1;
        else
            pHttpServer->m_hThreadList[i + 2] = hThread;
    }

    pHttpServer->m_sdListen = socket(AF_INET, SOCK_STREAM, 0);
    if(pHttpServer->m_sdListen == INVALID_SOCKET)
        return -1;

    BOOL bReuseAddr = TRUE;
    if(setsockopt(pHttpServer->m_sdListen, SOL_SOCKET, SO_REUSEADDR, (char *)
       &bReuseAddr, sizeof(bReuseAddr)) == SOCKET_ERROR)
        return -1;

    SOCKADDR_IN local;
    memset(&local, 0, sizeof(local));
    local.sin_family = AF_INET;
    local.sin_port = htons(g_pSvrOptions->m_nServerPort);
    local.sin_addr.S_un.S_addr = INADDR_ANY;
    if(bind(pHttpServer->m_sdListen, (SOCKADDR *) &local, sizeof(local)) ==
       SOCKET_ERROR)
        return -1;
```

```
    if(listen(pHttpServer->m_sdListen, 5) == SOCKET_ERROR)
        return -1;

    // 通知StartServer函数，监听线程已启动
    SetEvent(pHttpServer->m_evtThreadLanched);

    SOCKET sockAccept;
    LPCONN_CTX lpConnCtx;
    int nResult;
#ifdef _DEMO
    SOCKADDR_IN from;
#endif
    while(1){
        sockAccept = accept(pHttpServer->m_sdListen, NULL, NULL);
        if(sockAccept == INVALID_SOCKET){
            SetEvent(pHttpServer->m_evtSvrToStop);
            return -1;
        }
#ifdef _DEMO
        memset(&from, 0, sizeof(from));
        int fromlen = sizeof(from);
        getpeername(sockAccept, (SOCKADDR *) &from, &fromlen);
        AfxGetMainWnd()->PostMessage(WM_USER_CLIENT, from.sin_addr.s_addr,
           CLIENT_CONN_REQ);
#endif
        if(pHttpServer->m_bSvrPaused){
            closesocket(sockAccept);
#ifdef _DEMO
            AfxGetMainWnd()->PostMessage(WM_USER_CLIENT, from.sin_addr.s_
               addr, CLIENT_REJECT);
#endif
            continue;
        }


        lpConnCtx = pHttpServer->CreateConnCtx(sockAccept, pHttpServer->
           m_hIOCP);
        if(lpConnCtx == NULL){
            SetEvent(pHttpServer->m_evtSvrToStop);
            return -1;
        }
        else
            pHttpServer->ConnListAdd(lpConnCtx);
#ifdef _DEMO
        AfxGetMainWnd()->PostMessage(WM_USER_CLIENT, from.sin_addr.s_addr,
           CLIENT_ACCEPT);
```

```
#endif

          // 投递初始 I/O 操作
          nResult = WSARecv(sockAccept,
                                  &(lpConnCtx->pPerIOData->wbuf),
                                  1,
                                  NULL,
                                  &(lpConnCtx->pPerIOData->flags),
                                  &(lpConnCtx->pPerIOData->OverLapped),
                                  NULL);
          if((nResult == SOCKET_ERROR) && (WSAGetLastError() != ERROR_IO_
             PENDING)){
                pHttpServer->ConnListRemove(lpConnCtx);
                continue;
          }
     }

     return 0;
}

// 服务线程
DWORD WINAPI CHttpServer::ServiceThread(LPVOID pParam)
{
     CHttpServer *pHttpServer = (CHttpServer *) pParam;
     HANDLE hIOCP = pHttpServer->m_hIOCP;

     BOOL bSuccess = false;
     DWORD     dwIOSize;
     LPPER_IO_DATA lpPerIOData;
     LPOVERLAPPED pOverLapped;
     LPCONN_CTX lpConnCtx;
     int nResult;

     while(1){
          bSuccess = GetQueuedCompletionStatus(hIOCP, &dwIOSize, (LPDWORD)
            &lpConnCtx, &pOverLapped, INFINITE);
          if(lpConnCtx == NULL)
               return -1;

          lpPerIOData = (LPPER_IO_DATA) (pOverLapped);
          if(!bSuccess || (bSuccess && (dwIOSize == 0))){
               pHttpServer->ConnListRemove(lpConnCtx);
               continue;
          }

          switch(lpPerIOData->oper){
          case SVR_IO_WRITE:// 上一次操作是 Response - WSASend
               lpConnCtx->nAlreadyResponsed =+ dwIOSize;
```

```
            if(lpConnCtx->nAlreadyResponsed == lpConnCtx->szResponse.size()){// Response 完毕
                if(!lpConnCtx->bKeepAlive){// 不需要保持连接
                    pHttpServer->ConnListRemove(lpConnCtx);
                    continue;
                }
                else{// 需要保持连接，接收新的 Request
                    ZeroMemory(lpPerIOData, sizeof(PER_IO_DATA));
                    lpPerIOData->wbuf.buf = (char *) &(lpPerIOData->data);
                    lpPerIOData->wbuf.len = MAX_BUF_LEN;
                    lpPerIOData->oper = SVR_IO_READ;
                    lpPerIOData->flags = 0;

                    lpConnCtx->szResponse.erase(0, string::npos);
                    lpConnCtx->nAlreadyResponsed = 0;

                    nResult = WSARecv(lpConnCtx->sockAccept,
                        &(lpPerIOData->wbuf),
                        1,
                        NULL,
                        &(lpPerIOData->flags),
                        &(lpPerIOData->OverLapped),
                        NULL);
                    if(nResult == SOCKET_ERROR && WSAGetLastError() != ERROR_IO_PENDING){
                        pHttpServer->ConnListRemove(lpConnCtx);
                    }
                }
            }
            else{// 继续发送 Response
                lpPerIOData->oper = SVR_IO_WRITE;
                lpPerIOData->wbuf.buf = (char *) (lpConnCtx->szResponse.c_str() + lpConnCtx->nAlreadyResponsed);
                lpPerIOData->wbuf.len = lpConnCtx->szResponse.size() - lpConnCtx->nAlreadyResponsed;
                lpPerIOData->flags = 0;
                nResult = WSASend(lpConnCtx->sockAccept,
                                &(lpPerIOData->wbuf),
                                1,
                                NULL,
                                lpPerIOData->flags,
                                &(lpPerIOData->OverLapped),
                                NULL);
                if(nResult == SOCKET_ERROR && WSAGetLastError() != ERROR_IO_PENDING){
                    pHttpServer->ConnListRemove(lpConnCtx);
                }
            }
```

```
        break;
    case SVR_IO_READ: // 上一次操作是接收Request - WSARecv
        lpConnCtx->szRequest += string(lpPerIOData->data, dwIOSize);
        if(lpConnCtx->szRequest.size() > MAX_HTTP_REQUEST_LEN){
            pHttpServer->ConnListRemove(lpConnCtx);
            continue;
        }
        if(pHttpServer->IsRequestCompleted(lpConnCtx->szRequest)){
            // 已经得到完整的Request，作出Response
            pHttpServer->ProcessRequest(lpConnCtx->szRequest,
                                        lpConnCtx->szResponse,
                                        lpConnCtx->bKeepAlive);
            lpConnCtx->szRequest.erase(0, string::npos);
            lpPerIOData->oper = SVR_IO_WRITE;
            lpPerIOData->wbuf.buf = (char *) lpConnCtx->
                szResponse.c_str();
            lpPerIOData->wbuf.len = lpConnCtx->szResponse.size();
            lpPerIOData->flags = 0;
            nResult = WSASend(lpConnCtx->sockAccept,
                              &(lpPerIOData->wbuf),
                              1,
                              NULL,
                              lpPerIOData->flags,
                              &(lpPerIOData->OverLapped),
                              NULL);
            if(nResult == SOCKET_ERROR && WSAGetLastError() != ERROR_IO
              _PENDING){
                pHttpServer->ConnListRemove(lpConnCtx);
            }
        }
        else{// 否则继续read Request
            ZeroMemory(lpPerIOData, sizeof(PER_IO_DATA));
            lpPerIOData->oper = SVR_IO_READ;
            lpPerIOData->wbuf.buf = (char *) &(lpPerIOData->data);
            lpPerIOData->wbuf.len = MAX_BUF_LEN;
            lpPerIOData->flags = 0;

            nResult = WSARecv(lpConnCtx->sockAccept,
                    &(lpPerIOData->wbuf),
                    1,
                    NULL,
                    &(lpPerIOData->flags),
                    &(lpPerIOData->OverLapped),
                    NULL);
            if(nResult == SOCKET_ERROR && WSAGetLastError() != ERROR_IO
              _PENDING){
                pHttpServer->ConnListRemove(lpConnCtx);
            }
```

```
693                 }
694                 break;
695             default:
696                 ;
697             }
698         }

699         return 0;
700     }
//////////////////////////////////////////////////////////////////////////
```

第 12～16 行使用#define 定义了 CreateNewIoCompletionPort、AssociateWithIoCompletion-Port 两个宏定义函数，分别用于创建 I/O 完成端口和绑定完成端口。

第 19～31 行 CHttpServer 构造函数。

- 21～22 行，创建事件对象 m_evtThreadLanched、m_evtSvrToStop，这两个事件对象均为 Auto-Reset 类型，并且初始状态为未触发。
- 23 行，初始化临界区对象 m_CriticalSection。
- 24 行，调用 ResetAll 成员函数，复位内部变量。
- 25～28 行，获取系统的 CPU 处理器数目，将处理器数目×2 与 MAX_SERVICETHREAD_NUM 之间的较小值赋给成员变量 m_nSvcThreadNum。
- 30 行，调用 InitMimeTypes 成员函数，初始化 m_MimeTypes 值，建立文件后缀名与 MIME 资源类型之间的映射关系。参见代码 132～206 行。

第 32～37 行 CHttpServer 析构函数。关闭事件对象句柄 m_evtThreadLanched 和 m_evtSvrToStop，并且删除临界区对象 m_CriticalSection。

第 38～100 行 StartServer，public 成员函数，启动 MyWeb 服务。

- 40～47 行，判断服务器当前状态，如果已处于运行态，则函数直接返回 FALSE；如果本来处于暂停状态，那么将 m_bSvrPaused 设置为 FALSE，并且直接将 m_ServerState 设置为 SERVER_RUNNING，函数返回 TRUE；否则，进入下一步。
- 48～52 行，初始化 Winsock，并复位服务器。
- 53～68 行，启动管理线程。首先在 54～59 行创建管理线程，然后主线程在 60～67 行等待管理线程触发 m_evtThreadLanched 事件对象。如果在 WAIT4THREAD_MILLISECS 时间内该事件对象被触发，说明管理线程已正常启动；否则该线程未正常启动，进行一定的清理工作后，函数退出并返回 FALSE。在 68 行，将确定已正常启动的管理线程句柄保存至 m_hThreadList[0]。
- 70～97 行，启动监听线程。在 71 行，复位事件对象 m_evtThreadLanched。在 72 行，创建监听线程，根据返回值分两种情况处理：① 如果创建线程失败，那么触发 m_evtSvrToStop 事件（75 行），通知已处于运行中的管理线程终止服务，然后等待管理线程结束（77 行），如果在 WAIT4THREAD_MILLISECS*3 时间内管理线程未正常结束，那么调用 TerminateThread 函数将其强行终止（78 行）。在结束了管理线程之后，关闭其句柄，作清理工作后函数退出并返回 FALSE。② 线程创建成功，那么等待监听线程触发 m_evtThreadLanched 事件（85 行）。如果在

WAIT4THREAD_MILLISECS 时间内该事件对象未被触发，说明监听线程未能正常启动，和 Case I 进行同样的处理：通知管理线程终止服务，进行相关清理工作后函数退出并返回 FALSE。97 行，将已正常启动的监听线程句柄保存至 m_hThreadList[1]。

- 98～99 行，更改服务器状态值 m_ServerState 为 SERVER_RUNNING，函数正常返回 TRUE。

第 101～108 行 PauseServer，public 成员函数，暂停服务器。该函数所做工作非常简单，只需要将 m_bSvrPaused 成员变量赋值为 TRUE，并将服务器状态置为 SERVER_PAUSE。

第 109～125 行 StopServer，public 成员函数，停止服务。

- 111～112 行，如果服务器本来就处于 SERVER_STOP 状态，函数直接返回 FALSE，否则执行下一步。
- 113 行，触发事件对象 m_evtSvrToStop，通知管理线程终止服务器。
- 115 行，等待管理线程结束，时限为 WAIT4THREAD_MILLISECS * 3。
- 116～119 行，检查管理线程状态，如果该线程仍处于运行态，则将其强行终止。
- 121～122 行，调用 ResetAll 函数复位内部变量，调用 WSACleanup 结束 Winsock 的使用。
- 123 行，函数返回 TRUE。

第 126～129 行 GetServerState，public 成员函数，返回服务器的当前状态 m_ServerState。

第 132～206 行 InitMimeTypes，protected 成员函数，给 m_MimeTypes 成员变量赋值，建立文件后缀名与 MIME 资源类型之间的映射关系。

第 207～226 行 ResetAll，protected 成员函数，复位内部变量。

- 209～210 行，将服务器状态值 m_ServerState 置为 SERVER_STOP，将监听套接口 m_sdListen 置为 INVALID_SOCKET。
- 211～215 行，关闭运行线程句柄数组 m_hThreadList 中的非空句柄，并赋值为 NULL。
- 216～219 行，如果完成端口句柄 m_hIOCP 值非空，将其关闭，并置为 NULL。
- 220～221 行，调用 ConnListClear 成员函数清除连接信息链表 m_ptrConnCtxHead。
- 222～223 行，复位事件对象 m_evtThreadLanched 和 m_evtSvrToStop。
- 224～225 行，将 m_bSvrPaused 置为 FALSE，m_ptrConnCtxHead 置为 NULL。

第 229～238 行 IsRequestCompleted，protected 成员函数，用于判断 HTTP 请求是否已完整。如果请求信息 szRequest 中包含字符串“\r\n\r\n”，则认为该 HTTP 请求已完整，返回 TRUE，否则返回 FALSE。

第 239～354 行 ProcessRequest，protected 成员函数。需要注意的是该函数的 3 个输入参数：第一个参数是 szRequest，用于传入 HTTP 请求字符串；后两个分别是 szResponse 和 bKeepAlive 的引用，用于保存 szRequest 的处理结果。

在分析这一段代码之前，首先简单介绍 HTTP 协议的相关知识，有兴趣的读者可以查阅 RFC 1945. Hypertext Transfer Protocol -- HTTP/1.0 和 RFC 2068. Hypertext Transfer Protocol -- HTTP/1.1。

HTTP 协议的执行采用标准的 C/S 模式，客户端（通常是浏览器）发送用户请求 Request，服务器端解析 Request 并返回 Response。其中 Request 的定义如下：

```
Request = Request-Line
          *(General-Header | Request-Header| Entity-Header)
          <CRLF>
          [Message-Body]
Request-Line = Method<SP>Request-URI<SP>HTTP-Version<CRLF>
Method = "GET" | "HEAD"| "POST" | extension-method
Method = "OPTIONS" | "GET" | "HEAD" | "POST" | "PUT"
                   | "DELETE" | "TRACE" | extension-method
extension-method = token
```

其中<SP>表示空格，<CRLF>是回车换行。由上述定义可以总结出 HTTP 请求的基本格式是：

```
Method<SP> Request-URI<SP>……
```

Web 服务器在完成对 Request 解析和处理之后，返回 Response，Response 的定义如下：

```
Response = Status-Line
           *(General-Header | Response-Header | Entity-Header)
           <CRLF>
           [Message-Body]
Status-Line = HTTP-Version<SP>Status-Code<SP>Reason-Phrase<CRLF>
Status-Code   =     "200"   ; OK
                  | "201"   ; Created
                  | "202"   ; Accepted
                  | "204"   ; No Content
                  | "301"   ; Moved Permanently
                  | "302"   ; Moved Temporarily
                  | "304"   ; Not Modified
                  | "400"   ; Bad Request
                  | "401"   ; Unauthorized
                  | "403"   ; Forbidden
                  | "404"   ; Not Found
                  | "500"   ; Internal Server Error
                  | "501"   ; Not Implemented
                  | "502"   ; Bad Gateway
                  | "503"   ; Service Unavailable
| extension-code
```

我们对 HTTP 请求和响应的基本格式已经有一定的了解，下面对 ProcessRequest 成员函数进行分析：

- ❒ 255～286 行，解析 szRequest 请求字符串中所含的 Method，并读取所请求资源的 URI。
- ✧ 根据上面对 HTTP-Request 的分析可以知道，所有请求串都以“Method<SP> Request-URI<SP>……”的形式存在，因此在 256 行，首先获取 szRequest 串中空格所在的位置（变量 n）：如果 n 等于常量 string::npos，说明 szRequest 中不存在空格，那么该 HTTP 请求是无法识别的，将 szFileName 设置为 501 错误页面 ERROR501

（283～286 行）；否则，szRequest 中前 n 个字符即组成 HTTP-Request 中的 Method（变量 szMethod）。

- ✧ 由于 MyWeb 目前只支持"GET" Method，如果请求的 Method 不是"GET"，那么在 278～281 行将 szFileName 设置为 501 错误页面 ERROR501；否则，进行下一步对 Request-URI 的解析。
- ✧ 在 261 行，在去除 Method<SP>后剩下的 HTTP 请求串中查找空格（变量 n1），如果 n1 等于 string::npos，说明 Request-URI 无法解析，将 szFileName 设置为 501 错误页面 ERROR501（273～276 行）；否则 szRequest 从第 n+1 个字符开始的 n1-n-1 个字符组成 Request-URI（变量 szFileName）。
- ✧ 在 263～269 行，对 szFileName 作了简单的安全检查，以防止客户端请求的资源超出允许的目录范围：如果 szFileName 包含"..", szFileName 设置为 404 错误页面 ERROR404。需要指出的是，这种检查是必需的，但并不完备。
- ✧ 在 270～271 行，如果 szFileName 等于"/"，那么将 MyWeb 默认页面（g_pSvrOptions->m_szDefaultPage）的地址值赋给 szFileName。

- ❒ 287～292 行，确定是否要保存连接存活。在 szRequest 请求串中查找“Connection: Keep-Alive”，如果存在则认为需要保持连接，将 bKeepAlive 设置为 TRUE；否则 bKeepAlive 为 FALSE。
- ❒ 293～302 行，确定响应数据的资源类型。在 294～297 行，首先获取 szFileName 的后缀名 szFileExt；然后在 298～301 行，从 m_MimeTypes 对应表中查找 szFileExt 对应的 MIME 资源类型（变量 szContentType）。
- ❒ 303～309 行，获取当前 GMT 格式的日期、时间（变量 szDT）。
- ❒ 310～353 行，组装 HTTP-Response（变量 szResponse）。

- ✧ 311～313 行，根据 MyWeb 服务器的根目录路径（g_pSvrOptions->m_szHomeDir）和文件名（szFileName）组装被请求资源的完整路径 szFullPathName，并打开文件。
- ✧ 314～326 行，如果 szFullPathName 对应文件存在，那么将文件数据读取至 pBuf，并按 HTTP/1.0 规范组装 pResponseHeader。然后在 348～352 行，将 pBuf 数据附在 pResponseHeader 之后组成 szResponse。
- ✧ 327～353 行，如果 szFullPathName 对应文件不存在，那么选择打开 404 错误页面 ERROR404，即\\E404.html，同样读取文件内容（变量 szNotFoundMessage）。无论 ERROR404 文件是否存在，都按 HTTP/1.0 规范组装 pResponseHeader，并将 bKeepAlive 设定为 FALSE。最后在 348～352 行直接将 pResponseHeader 的值赋给 szResponse。

- ❒ 357～387 行，CreateConnCtx，protected 成员函数。如前所述，主要完成两个工作：一是将套接口 sockAccept 与完成端口 hIOCP 绑定，二是为新到来的客户端连接创建 CONN_CTX 数据。从 359～379 行，创建 CONN_CTX 类对象 lpConnCtx 并赋初值；从 380～385 行，将 sockAccept 与 hIOCP 绑定。在上述过程中，如果有错误发生，返回 NULL，否则返回 lpConnCtx。
- ❒ 388～406 行，ConnListAdd，protected 成员函数，将连接信息 lpConnCtx 加入到 CHttpServer 模块维持的全局连接链表。属于普通的双向链表操作，惟一需要注意的

是，由于可能有多个线程同时对该双向链表进行操作，该函数使用了 m_CriticalSection 临界区变量来进行线程的互斥管理。

- 407～447 行，ConnListRemove，protected 成员函数，将连接信息 lpConnCtx 从 CHttpServer 模块维持的全局连接链表中删除，该函数一般在连接结束的时候调用。与上面的 ConnListAdd 成员函数一样，ConnListRemove 完成基本的双向链表操作，并且实现了线程之间互斥的资源访问管理。下面着重介绍 432～438 行代码。
- 432 行，使用条件编译，如果在项目中定义了_DEMO，那么执行 433～437 行的代码，否则直接跳过。在本项目的 Preprocessor Definitions 中加入了_DEMO，如图 22.7 所示。

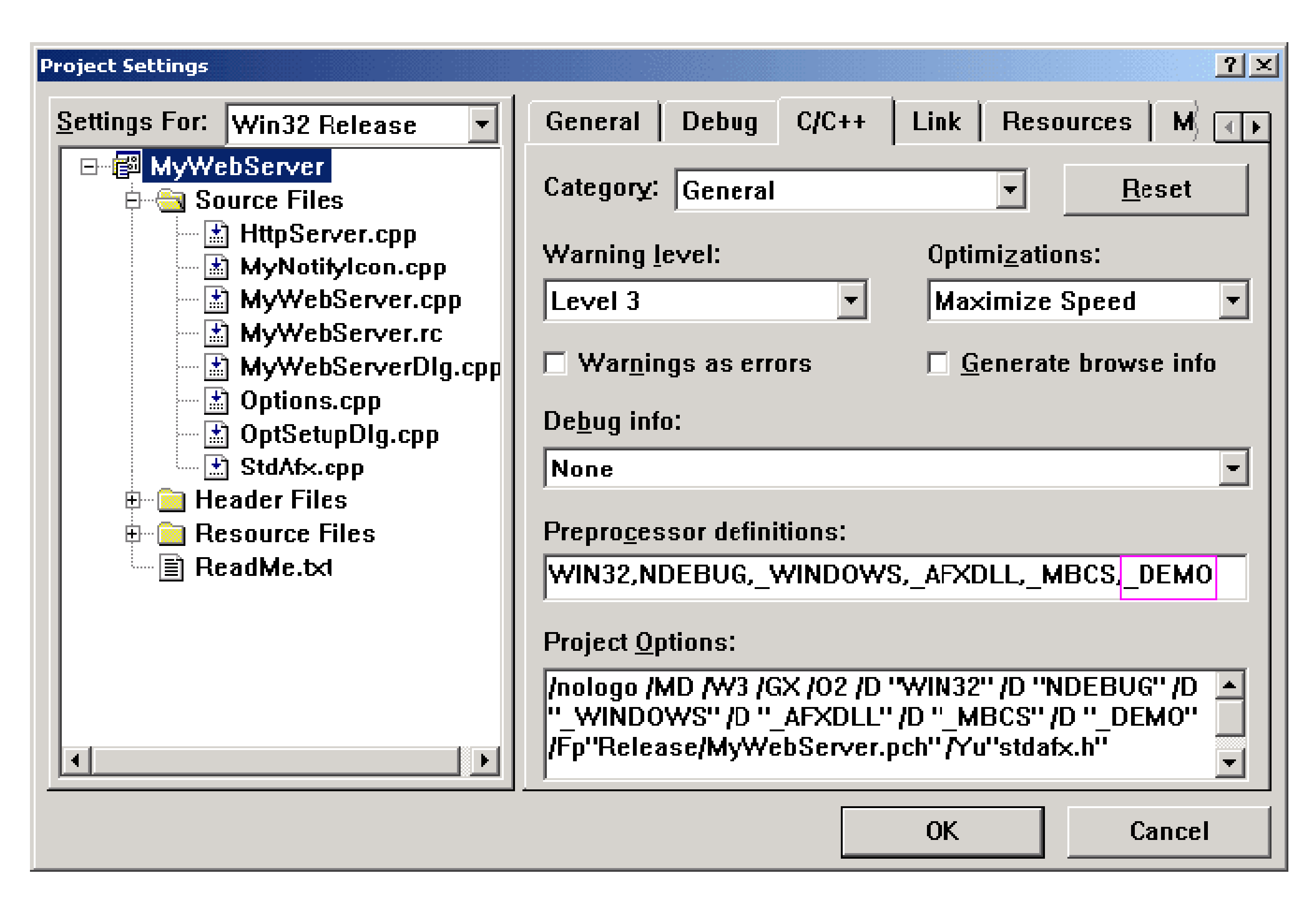


图 22.7 项目设置

- 433～436 行，调用 getpeername 函数获取 lpConnCtx 对应的连接的客户端的 IP 地址。
- 437 行，向主窗体发送消息 WM_USER_CLIENT，两个参数分别是客户端地址 from.sin_addr.s_addr 和自定义客户消息类型 CLIENT_DISCONNECT。
- 438 行，结束_DEMO 条件编译。
- 448～460 行，ConnListClear，protected 成员函数，调用 ConnListRemove 成员函数关闭当前的所有连接并清除连接信息链表。
- 463～700 行，CHttpServer 类的 3 个静态成员函数：AdminThread、ListenThread 和 ServiceThread，分别是服务器的管理线程函数、监听线程函数和服务线程函数。

事实上，在整个 MyWeb 服务器系统包含了 4 类线程，除了上述 3 种线程外，还包括系统的主线程，对应于用户 GUI 界面。在主线程中，创建了 CHttpServer 类的一个实例，用户通过单击某些菜单可以调用该实例的 StartServer/PauseServer/StopServer 等 public 成员函数。图 22.8 展现了这 4 类线程之间的同步关系，虚线箭头表示启动（调用）和触发，下面对这些线程之间的关系进行详细分析。

主线程
（StartServer）
调用 StartServer
启动 AdminThread
m_evtThreadLanched 事件对象
是否在一定时间内被触发?
否
是
启动 ListenThread
m_evtThreadLanched 事件对象
是否在一定时间内被触发?
否
是
触发 m_evtSvrToStop
AdminThread
是否在一定时间
内结束?
是
否
StartServer结束
强制结束
AdminThread
管理线程
启动 AdminThread
触发 m_evtThreadLanched
无限期等待
m_evtSvrToStop 事件
关闭监听套接口，
投递特殊 I/O 完成信息包
监听 /服务线程
是否在一定时间
内结束?
是
否
强制关闭所有
监听 /服务线程
AdminThread结束
主线程
（StopServer）
调用 StopServer
触发 m_evtSvrToStop
AdminThread
是否在一定时间
内结束?
是
否
强制结束 AdminThread
StopServer结束
监听线程
启动 ListenThread
启动多个 ServiceThread
前期工作
是否成功?
否
是
触发 m_evtThreadLanched
accept是否成功?
否
是
循环
投递初始 I/O 操作
触发 m_evtSvrToStop
ListenThread结束
服务线程
启动 ServiceThread
是否接收到特殊
完成信息包?
是
否
循环
I/O 处理
ServiceThread结束

图 22.8　线程间的关系



1．从用户操作的角度来分析线程之间的调用和相互触发关系

（1）启动服务器

CHttpServer 类的成员函数 StartServer 在主线程中被调用，该函数首先启动管理线程 AdminThread，然后等待 AdminThread 触发事件 m_evtThreadLanched。如果 m_evtThreadLanched 在一定时间内被触发，那么说明 AdminThread 已正常启动，StartServer 进入下一步，否则 StartServer 函数退出。

随后，StartServer 启动监听线程 ListenThread，同样等待 ListenThread 触发 m_evtThreadLanched。如果 m_evtThreadLanched 在一定时间内被触发，那么说明 ListenThread 已正常启动，StartServer 函数结束；否则说明 ListenThread 未正常启动，StartServer 将触发 m_evtSvrToStop，通知管理线程结束服务器。正常情况下，管理线程在结束监听线程和服务线程后会自动结束，但是某些情况下，如果管理线程未能正常结束，那么 StartServer 会在退出之前强制关闭管理线程。

（2）停止服务器

StopServer 函数也在主线程中被调用，该函数触发 m_evtSvrToStop 事件通知管理线程结束服务器。如果管理线程在一定时间内自动结束，StopServer 退出；否则强制结束管理线程后再退出。

2．从线程的启动和结束角度分析线程之间的同步

（1）管理线程

管理线程 AdminThread 由主线程调用 StartServer 函数启动。在启动之后，AdminThread 立即触发 m_evtThreadLanched 事件通知 StartServer 本线程已启动，随后线程阻塞在对 m_evtSvrToStop 事件的等待上。

在 3 种情况下，m_evtSvrToStop 事件会被触发：

① 调用 StartServer 函数时，ListenThread 未能正常启动。

② ListenThread 运行过程中 accept 出错。

③ 调用 StopServer 函数，停止服务器。

在 m_evtSvrToStop 事件被触发后，管理线程采用两种手段关闭其他线程：

① 关闭服务器监听套接口，使得 ListenThread 的 accept 调用出错，从而退出。

② 投递与系统中服务线程数目相等的特殊完成信息包，通知 ServiceThread 退出。

随后，管理线程检测监听线程和服务线程在一段时间后是否已正常结束，如果没有结束则强制关闭。完成所有这些工作后，管理线程结束。

（2）监听线程

监听线程 ListenThread 由主线程调用 StartServer 函数而启动。在被启动之前，AdminThread 已成功启动。

ListenThread 首先启动若干个 ServiceThread，然后进行一些监听服务的前期准备工作。在这个过程中，如果有错误发生，那么 ListenThread 直接终止，从而导致 StartServer 等待 m_evtThreadLanched 超时；如果没有错误发生，则触发 m_evtThreadLanched 事件通知 StartServer 本线程已成功启动。接下来的是一个服务循环：ListenThread 调用 accept 函数接受外来的连接请求，并投递初始的 I/O 操作。如果在 accept 时出错，认为或者是发生了严重的软件故障，或者是 AdminThread 通知 ListenThread 结束服务。无论具体是哪一种情况，

ListenThread 都触发 m_evtSvrToStop 事件，然后终止线程。

（3）服务线程

服务线程 ServiceThread 由监听线程启动，用于处理 I/O 数据，包括读取、分析客户端的 HTTP-Request，并给出 Response。在系统中会有多个 ServiceThread 同时工作，每个 ServiceThread 都处于一个循环中：GetQueuedCompletionStatus－投递读/写操作请求－GetQueuedCompletionStatus……。只有在 GetQueuedCompletionStatus 获取到特殊的完成信息（完成键为 NULL，只可能由管理线程发送）时，服务线程才终止服务。

下面对管理线程、监听线程和服务线程的代码进行分析：

❐ 463～494 行，AdminThread，管理线程函数。

✧ 466 行，获取 CHttpServer 类实例的指针。在介绍 HttpServer.h 文件时提到过 AdminThread 是 CHttpServer 类的静态成员函数，因此没有 this 指针，也就无法调用类的非静态成员函数。这里提供了一个方法来解决这个问题：在 StartServer 启动 AdminThread 线程时，将 this 指针作为线程函数的参数 pParam 传递给 AdminThread，在 AdminThread 中将 pParam 强制转化为 CHttpServer 类指针后，即可读取/调用类的成员变量和函数。此方法同样被应用于监听线程 ListenThread。

✧ 467～468 行，触发 m_evtThreadLanched 事件对象，通知 StartServer 函数 AdminThread 已启动。

✧ 473 行，阻塞至 m_evtSvrToStop 事件对象被触发。

✧ 475～476 行，关闭服务器监听套接口。

✧ 477～480 行，投递 m_nSvcThreadNum 个特殊完成信息包，通知服务线程终止。

✧ 481～485 行，等待监听线程和所有服务线程结束，等待时间为 WAIT4THREAD_MILLISECS * 2。

✧ 486～491 行，检查监听线程和所有服务线程的状态，如果仍然处于运行中，则将其强行终止。

✧ 492～493 行，调用 ResetAll 成员函数，确保所有变量均被复位，所有事件对象都处于未触发状态。

❐ 495～581 行，ListenThread，监听线程函数。

✧ 498 行，获取 CHttpServer 类实例的指针。

✧ 499～502 行，创建完成端口。如果创建失败，ListenThread 直接退出，从而导致 StartServer 在等待 m_evtThreadLanched 事件触发时超时，并进一步导致 m_evtSvrToStop 事件被触发。

✧ 503～510 行，创建 m_nSvcThreadNum 个服务线程，同样如果出错则立即退出线程。

✧ 512～527 行，创建监听端口，设置 SO_REUSEADDR 选项为 TRUE，绑定本地端口 g_pSvrOptions->m_nServerPort，并调用 listen 函数进行监听。在此过程中，如果有错误发生则直接结束 ListenThread。

✧ 528～529 行，触发 m_evtThreadLanched 事件对象，通知 StartServer 监听线程已正常启动。

✧ 530～579 行，循环接受外来连接，并投递初始 I/O 操作请求。

✧ 537～541 行，接受外来连接请求并返回 sockAccept（537），如果 accept 出错（538），

那么触发事件对象 m_evtSvrToStop 并退出线程（539～560），否则进入下一步。

✧ 542～547 行，条件编译代码，如果定义了 _DEMO，那么向主窗口发送消息 WM_USER_CLIENT，消息参数分别是客户端 IP 地址 from.sin_addr.s_addr 和消息子类型 CLIENT_CONN_REQ（客户端请求连接）。

✧ 548～554 行，如果服务器处于暂停状态（m_bSvrPaused 为 TRUE），关闭该客户端的连接 sockAccept，然后调用 continue 函数进入下一循环。同时，如果定义了 _DEMO，则向主窗口发送消息 WM_USER_CLIENT，消息参数是客户端 IP 地址 from.sin_addr.s_addr 和消息子类型 CLIENT_REJECT（拒绝客户端连接请求）。

✧ 556～562 行，为该新建连接（sockAccept）调用 CreateConnCtx 成员函数创建连接信息数据（lpConnCtx，在 CreateConnCtx 成员函数中还完成了将 sockAccept 与完成端口 m_hIOCP 绑定的工作）。在此过程中，如果有错误发生（lpConnCtx 为 NULL），那么触发事件对象 m_evtSvrToStop 并退出线程，否则将 lpConnCtx 加入到连接信息链表中。

✧ 563～565 行，向主窗口发送消息 WM_USER_CLIENT，消息参数是客户端 IP 地址 from.sin_addr.s_addr 和消息子类型 CLIENT_ACCEPT（接受客户端连接请求）。

✧ 567～578 行，调用 WSARecv 函数在套接口 sockAccept 上投递初始的读操作请求。如果 WSARecv 返回非 ERROR_IO_PENDING 错误，那么关闭连接并删除连接信息，然后进入下一循环。

❒ 582～700 行，ServiceThread，服务线程函数。

✧ 585 行，获取 CHttpServer 类实例的指针。

✧ 594～698 行，服务循环，获取 I/O 完成信息包，根据 I/O 类型和数据，组装、解析客户端的 HTTP-Request，或者发送服务器 Response。

✧ 595 行，调用 GetQueuedCompletionStatus 函数（返回值为 bSuccess）获取 I/O 完成信息包。

✧ 596～597 行，如果 lpConnCtx 为 NULL，说明线程收到了完成键为 NULL 的特殊完成信息包，服务线程终止。

✧ 598～602 行，如果 bSuccess 为 FALSE 或者 bSuccess 为 TRUE，但是 I/O 数据量 dwIOSize 为 0，那么说明发生了错误或者客户端断开了连接，服务线程将 lpConnCtx 从连接信息链表中删除，并直接进入下一个循环。

✧ 603～697 行，根据完成信息包中的 I/O 类型 lpPerIOData->oper，分两种情况进行处理：

✧ 604～649 行，SVR_IO_WRITE，说明服务器完成的是写操作（WSASend），也就是进行了 Response。

① 605 行，累加，计算服务器对客户端的当前 HTTP-Request 的 Response 一共已发送了多少数据 lpConnCtx->nAlreadyResponsed。按该值与 Response 长度的大小关系，分②、③两种情况处理。

② 606～632 行，已经发送数据量等于 Response 的长度，说明 Response 完毕。下一步同样分两种情况进行处理：

❍ 607～610 行，lpConnCtx->bKeepAlive 为 FALSE，说明不需要进行连接保持，服务

器关闭连接并将连接信息从连接链表中删除。

❍ 611～631 行，lpConnCtx->bKeepAlive 为 TRUE，说明需要进行连接保持，在复位并设置相关参数和缓冲区后，服务器投递读操作请求。

③ 633～648 行，已经发送数据量小于 Response 的长度，说明 Response 未结束，正确设置缓冲区偏移量后，服务器继续发送数据。

✧ 650～694 行，SVR_IO_READ，说明服务器完成的是读操作（WSARecv），也就是读取了 Request。

① 651～655 行，将新读取的数据添加到连接信息 lpConnCtx 中的客户端 HTTP 请求缓冲区 szRequest 中。如果此时的 szRequest 长度大于最长 HTTP 请求长度值 MAX_HTTP _REQUEST_LEN，就认为该数据有问题，服务器关闭对应的客户端连接并从连接链表中将其删除。

② 656～693 行，调用 IsRequestCompleted 成员函数，检查接收到的 HTTP-Request 是否已完整，按 IsRequestCompleted 的返回值分两种情况处理：

❍ 657～676 行，TRUE，说明服务器已接收到完整的 HTTP 请求。调用 ProcessRequest 成员函数处理 lpConnCtx->szRequest，并获得输出 lpConnCtx->szResponse 和 lpConnCtx->bKeepAlive 后，发送 lpConnCtx->szResponse。

❍ 677～693 行，FALSE，说明服务器未接收完毕 HTTP 请求，继续接收请求数据。

22.2.5　CMyWebServerDlg 类

CMyWebServerDlg 是最后一个较为重要的类，该类由系统自动生成，对应于 MyWeb 的主对话框，如图 22.9 所示。基本上所有的操作都围绕系统消息或者用户消息进行，所有的函数都是消息处理函数。

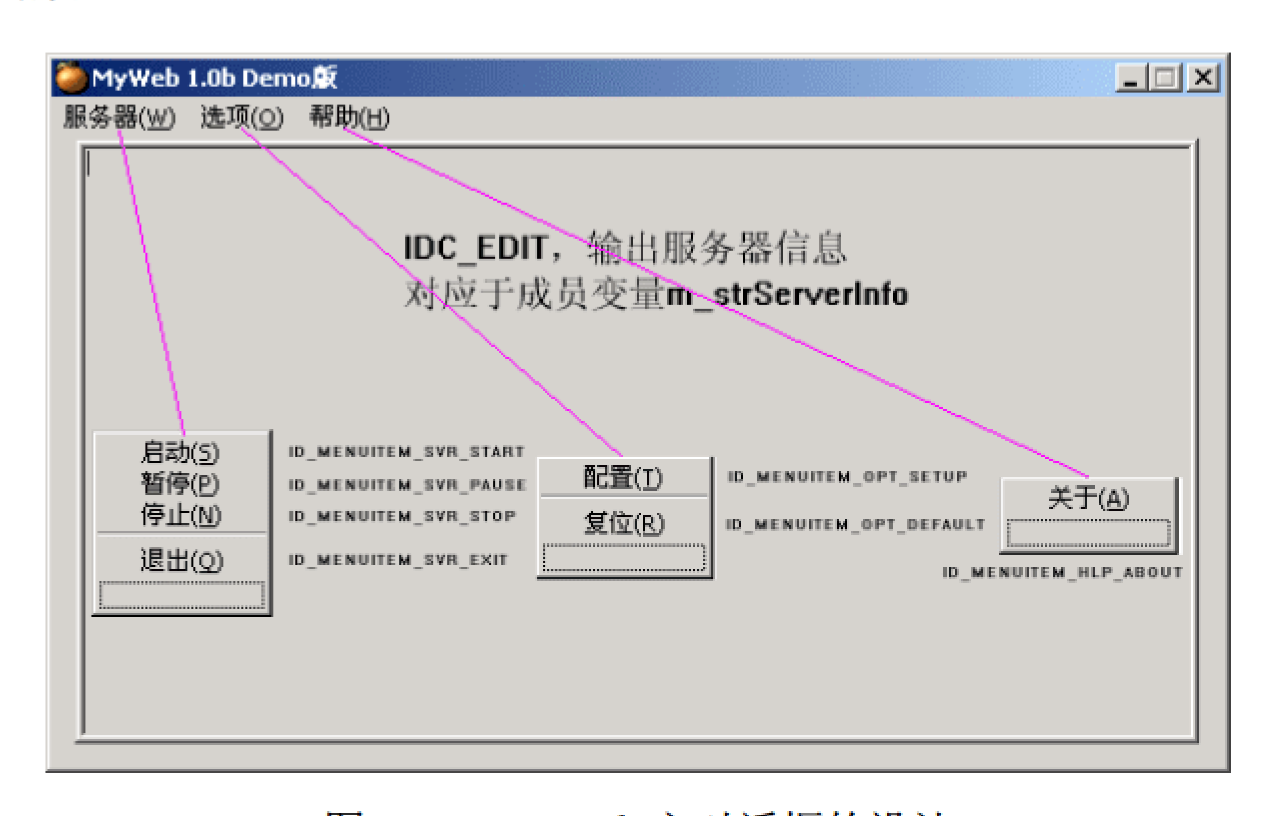


图 22.9　MyWeb 主对话框的设计

下面首先看看该类的接口声明：

```
////////////////////////////////////////////////////////////////////////////////
// MyWebServerDlg.h: CMyWebServerDlg 类的接口声明
// Create: 2004-03-23  LastModify: 2004-04-05
////////////////////////////////////////////////////////////////////////////////
```

```
#ifndef(AFX_MYWEBSERVERDLG_H__30C0629F_292F_4206_9934_773E43D351C7__INCLUDED_)
#define AFX_MYWEBSERVERDLG_H__30C0629F_292F_4206_9934_773E43D351C7__INCLUDED_

#include "Options.h"     // Added by ClassView
#include "MyNotifyIcon.h"     // Added by ClassView
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define WM_USER_SHOWWND             WM_USER+100
#define WM_USER_NOTIFYICON           WM_USER+101
#define TRAY_ICON_ID                  1234

/////////////////////////////////////////////////////////////////////////////
// CMyWebServerDlg dialog

class CMyWebServerDlg : public CDialog
{
// Construction
public:
    void AppendSvrInfo(const char *strNewInfo);
    CMyWebServerDlg(CWnd* pParent = NULL);     // standard constructor

// Dialog Data
    //{{AFX_DATA(CMyWebServerDlg)
    enum { IDD = IDD_MYWEBSERVER_DIALOG };
    CString     m_strServerInfo;
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMyWebServerDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);     // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    CMyNotifyIcon *m_pNotifyIcon;
    void UpdateMenuState();
    HICON m_hIcon;

    void PopMenu(WPARAM wParam, LPARAM lParam);
    // Generated message map functions
    //{{AFX_MSG(CMyWebServerDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
```

```
40        afx_msg HCURSOR OnQueryDragIcon();
41        afx_msg void OnHlpAbout();
42        afx_msg void OnOptSetup();
43        afx_msg void OnOptDefault();
44        afx_msg BOOL OnSvrStart();
45        afx_msg BOOL OnSvrPause();
46        afx_msg BOOL OnSvrStop();
47        afx_msg void OnSvrExit();
48        afx_msg void OnClose();
49        afx_msg void OnShowWnd();
50        afx_msg void OnClientMsg(WPARAM wParam, LPARAM lParam);
51        afx_msg LRESULT OnNotifyMsg(WPARAM wParam, LPARAM lParam);
52        //}}AFX_MSG
53        DECLARE_MESSAGE_MAP()
54    };

55    //{{AFX_INSERT_LOCATION}}
56    // Microsoft Visual C++ will insert additional declarations immediately before
        the previous line.

57    #endif
///////////////////////////////////////////////////////////////////////////////
```

在上面的源码中，黑体部分由用户手动编写，其余代码基本上都由系统根据用户定义的消息-函数对应关系自动添加。

第 8～9 行定义了两个用户自定义消息常量。

第 10 行定义了任务栏图标的 ID。

第 17 行 AppendSvrInfo，public 成员函数，用于向主对话框的服务器信息框（对话框的中间部分）输出信息。见下面对 m_strServerInfo 成员变量的解释。

第 22 行 m_strServerInfo，CString 型的 public 成员变量，对应于 EDIT 框 IDC_EDIT，如图 22.9 所示。

第 31 行声明了 CMyNotifyIcon 类指针 m_pNotifyIcon。

下面给出 CMyWebServerDlg 中各个成员消息函数与用户/系统消息的对应关系，如表 22.1 所示。

表22.1　消息与消息处理函数

函数名称	消息 ID	描　述
OnSvrStart	ID_MENUITEM_SVR_START	响应菜单消息，启动 MyWeb 服务器
OnSvrPause	ID_MENUITEM_SVR_PAUSE	响应菜单消息，暂停 MyWeb 服务器
OnSvrStop	ID_MENUITEM_SVR_STOP	响应菜单消息，停止 MyWeb 服务器
OnSvrExit	ID_MENUITEM_SVR_EXIT	响应菜单消息，退出 MyWeb 服务器
OnOptSetup	ID_MENUITEM_OPT_SETUP	响应菜单消息，进行服务器选项配置
OnOptDefault	ID_MENUITEM_OPT_DEFAULT	响应菜单消息，将服务器选项设置复位

OnHlpAbout	ID_MENUITEM_HLP_ABOUT	响应菜单消息，显示 About 对话框

续表

函数名称	消息 ID	描　　述
OnNotifyMsg	WM_USER_NOTIFYICON	响应用户自定义消息，处理任务栏图标的鼠标事件
OnShowWnd	WM_USER_SHOWWND	响应任务栏图标弹出菜单中<显示主窗口>选项消息，显示主对话框
OnClientMsg	WM_USER_CLIENT	响应用户自定义消息，处理 CHttpServer 发送的服务器相关的消息

下面是 CMyWebServerDlg 类的实现代码，同样，黑体部分表示由用户输入的重要代码，其他为系统自动生成。

```
///////////////////////////////////////////////////////////////////////////////
// MyWebServerDlg.cpp: CMyWebServerDlg 类的实现
// Create: 2004-03-23  LastModify: 2004-04-05
///////////////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "MyWebServer.h"
#include "MyWebServerDlg.h"
#include "Options.h"
#include "OptSetupDlg.h"
#include "HttpServer.h"
#include "MyNotifyIcon.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAboutDlg)
    protected:
```

```
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
	//}}AFX_VIRTUAL

// Implementation
protected:
	//{{AFX_MSG(CAboutDlg)
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
	//{{AFX_DATA_INIT(CAboutDlg)
	//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CAboutDlg)
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
	//{{AFX_MSG_MAP(CAboutDlg)
		// No message handlers
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMyWebServerDlg dialog

CMyWebServerDlg::CMyWebServerDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CMyWebServerDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CMyWebServerDlg)
	m_strServerInfo = _T("");
	//}}AFX_DATA_INIT
	// Note that LoadIcon does not require a subsequent DestroyIcon in Win32
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CMyWebServerDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CMyWebServerDlg)
	DDX_Text(pDX, IDC_EDIT, m_strServerInfo);
	//}}AFX_DATA_MAP
```

```
}

BEGIN_MESSAGE_MAP(CMyWebServerDlg, CDialog)
    //{{AFX_MSG_MAP(CMyWebServerDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_COMMAND(ID_MENUITEM_HLP_ABOUT, OnHlpAbout)
    ON_COMMAND(ID_MENUITEM_OPT_SETUP, OnOptSetup)
    ON_COMMAND(ID_MENUITEM_OPT_DEFAULT, OnOptDefault)
    ON_COMMAND(ID_MENUITEM_SVR_START, OnSvrStart)
    ON_COMMAND(ID_MENUITEM_SVR_PAUSE, OnSvrPause)
    ON_COMMAND(ID_MENUITEM_SVR_STOP, OnSvrStop)
    ON_COMMAND(ID_MENUITEM_SVR_EXIT, OnSvrExit)
    ON_WM_CLOSE()
    ON_COMMAND(WM_USER_SHOWWND, OnShowWnd)
    ON_MESSAGE(WM_USER_CLIENT, OnClientMsg)
    ON_MESSAGE(WM_USER_NOTIFYICON, OnNotifyMsg)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMyWebServerDlg message handlers

BOOL CMyWebServerDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    // Add "About..." menu item to system menu.
    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);             // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon
```

```
    // TODO: Add extra initialization here
    UpdateMenuState();
    m_pNotifyIcon = new CMyNotifyIcon(m_hWnd, WM_USER_NOTIFYICON, TRAY
      _ICON_ID);
    m_pNotifyIcon->AddIcon(IDI_ICON_STOP, "STOP");

    return TRUE;  // return TRUE  unless you set the focus to a control
}
void CMyWebServerDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
        if((nID & 0xFFF0) == SC_MINIMIZE)
            ShowWindow(SW_HIDE);
        else
            CDialog::OnSysCommand(nID, lParam);
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CMyWebServerDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
```

```
            CDialog::OnPaint();
        }
}

// The system calls this to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CMyWebServerDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}

void CMyWebServerDlg::OnHlpAbout()
{
    // TODO: Add your command handler code here
    CAboutDlg dlg;
    dlg.DoModal();
}

void CMyWebServerDlg::OnOptSetup()
{
    // TODO: Add your command handler code here
    COptSetupDlg dlg;
    if(dlg.DoModal() == IDOK){
        if(g_pHttpServer->GetServerState() == SERVER_RUNNING){
            if(MessageBox("是否重新启动服务器?", "重启服务器确认", MB_ICONQUESTION
               | MB_YESNO) == IDYES){
                OnSvrStop();
                OnSvrStart();
            }
        }
    }
}

void CMyWebServerDlg::OnOptDefault()
{
    // TODO: Add your command handler code here
    if(MessageBox("如果选择复位，您所作的服务器选项设置都将丢失，是否确认?", "服务器选
       项复位确认", MB_ICONQUESTION | MB_YESNO) == IDYES){
        g_pSvrOptions->LoadDefOptions();
        if(g_pHttpServer->GetServerState() == SERVER_RUNNING){
            if(MessageBox("是否重新启动服务器?", "重启服务器确认", MB_ICONQUESTION
               | MB_YESNO) == IDYES){
                OnSvrStop();
                OnSvrStart();
            }
        }
    }
```

```
    return;
}
BOOL CMyWebServerDlg::OnSvrStart()
{
    // TODO: Add your command handler code here
    if(!g_pHttpServer->StartServer()){
        AppendSvrInfo("MyWeb 启动服务失败,请稍后重试");
        return FALSE;
    }
    UpdateMenuState();
    UpdateData();
    CString info;
    info.Format("MyWeb 服务已启动. 端口: %d 目录: %s 默认页: %s",
        g_pSvrOptions->m_nServerPort,
        g_pSvrOptions->m_szHomeDir,
        g_pSvrOptions->m_szDefaultPage
        );
    AppendSvrInfo(info);
    m_pNotifyIcon->ChangeIcon(IDI_ICON_RUNNING, "Running");
    return TRUE;
}
BOOL CMyWebServerDlg::OnSvrPause()
{
    // TODO: Add your command handler code here
    if(!g_pHttpServer->PauseServer()){
        AppendSvrInfo("MyWeb 暂停服务失败,请稍后重试");
        return FALSE;
    }
    UpdateMenuState();
    AppendSvrInfo("MyWeb 暂停服务");
    m_pNotifyIcon->ChangeIcon(IDI_ICON_PAUSE, "Pause");

    return TRUE;
}
BOOL CMyWebServerDlg::OnSvrStop()
{
    // TODO: Add your command handler code here
    if(!g_pHttpServer->StopServer()){
        AppendSvrInfo("MyWeb 停止服务失败, 请稍后重试");
        return FALSE;
    }
    UpdateMenuState();
    AppendSvrInfo("MyWeb 停止服务");
    m_pNotifyIcon->ChangeIcon(IDI_ICON_STOP, "Stop");
```

```
    return TRUE;
}

void CMyWebServerDlg::OnSvrExit()
{
    // TODO: Add your command handler code here
    if(g_pHttpServer->GetServerState() != SERVER_STOP){
        if(MessageBox("服务器运行中，是否确认退出?", "退出确认", MB_ICONQUESTION |
          MB_YESNO) == IDNO)
            return;
    }
    OnSvrStop();

    m_pNotifyIcon->DelIcon();
    delete m_pNotifyIcon;

    CDialog::OnCancel();
}

void CMyWebServerDlg::OnClose()
{
    // TODO: Add your message handler code here and/or call default
    OnSvrExit();
}


void CMyWebServerDlg::UpdateMenuState()
{
    ServerState state = g_pHttpServer->GetServerState();
    CMenu *pMenu = GetMenu();
    ASSERT(pMenu != NULL);
    switch(state){
    case SERVER_STOP:
        pMenu->EnableMenuItem(ID_MENUITEM_SVR_START, MF_ENABLED);
        pMenu->EnableMenuItem(ID_MENUITEM_SVR_PAUSE,MF_DISABLED|MF_GRAYED);
        pMenu->EnableMenuItem(ID_MENUITEM_SVR_STOP, MF_DISABLED | MF_GRAYED);
        break;
    case SERVER_PAUSE:
        pMenu->EnableMenuItem(ID_MENUITEM_SVR_START, MF_ENABLED);
        pMenu->EnableMenuItem(ID_MENUITEM_SVR_PAUSE,MF_DISABLED|MF_GRAYED);
        pMenu->EnableMenuItem(ID_MENUITEM_SVR_STOP, MF_ENABLED);
        break;
    case SERVER_RUNNING:
        pMenu->EnableMenuItem(ID_MENUITEM_SVR_START,MF_DISABLED|MF_GRAYED);
        pMenu->EnableMenuItem(ID_MENUITEM_SVR_PAUSE, MF_ENABLED);
        pMenu->EnableMenuItem(ID_MENUITEM_SVR_STOP, MF_ENABLED);
        break;
```

```
    }
}

void CMyWebServerDlg::AppendSvrInfo(const char *strNewInfo)
{
    time_t ts;
    struct tm *local;
    time(&ts);
    local = localtime(&ts);
    CString tmp;
    tmp.Format("%s: %s\r\n", asctime(local), strNewInfo);
    m_strServerInfo += tmp;

    UpdateData(FALSE);
}

void CMyWebServerDlg::OnClientMsg(WPARAM wParam, LPARAM lParam)
{
    IN_ADDR client;
    client.s_addr = wParam;
    char* ClientMsg[] = {"请求连接", "连接被接受", "连接被拒绝", "连接关闭", "发生错
      误，连接结束"};
    CString info;
    info.Format("From IP 地址 %s, %s", inet_ntoa(client), ClientMsg[lParam]);
    AppendSvrInfo(info.operator LPCTSTR());
}

void CMyWebServerDlg::OnShowWnd()
{
    ShowWindow(SW_SHOW);
}

LRESULT CMyWebServerDlg::OnNotifyMsg(WPARAM wParam, LPARAM lParam)
{
    switch(lParam){
    case WM_LBUTTONDBLCLK:
        ShowWindow(SW_SHOW);
        break;
    case WM_RBUTTONDOWN:
        PopMenu(wParam, lParam);
        break;
    default:
        ;
    }

    return 0;
}
```

```
void CMyWebServerDlg::PopMenu(WPARAM wParam, LPARAM lParam)
{
    if(wParam == TRAY_ICON_ID && lParam == WM_RBUTTONDOWN)
    {
        CMenu ContextMenu;
        CPoint CursorPos;
        GetCursorPos(&CursorPos);
        ContextMenu.CreatePopupMenu();
        ServerState state = g_pHttpServer->GetServerState();
        ContextMenu.AppendMenu(MF_STRING, WM_USER_SHOWWND, _T("显示主窗口"));
        ContextMenu.SetDefaultItem(0, TRUE);
        ContextMenu.AppendMenu(MF_STRING, ID_MENUITEM_HLP_ABOUT, _T("关于MyWeb"));
        ContextMenu.AppendMenu(MF_STRING, ID_MENUITEM_OPT_SETUP, _T("配置服务器"));
        ContextMenu.AppendMenu(MF_SEPARATOR);

        switch(state){
        case SERVER_STOP:
            ContextMenu.AppendMenu(MF_STRING, ID_MENUITEM_SVR_START, _T("启动MyWeb"));
            ContextMenu.AppendMenu(MF_GRAYED, ID_MENUITEM_SVR_PAUSE, _T("暂停MyWeb"));
            ContextMenu.AppendMenu(MF_GRAYED, ID_MENUITEM_SVR_STOP, _T("停止MyWeb"));
            break;
        case SERVER_PAUSE:
            ContextMenu.AppendMenu(MF_STRING, ID_MENUITEM_SVR_START, _T("启动MyWeb"));
            ContextMenu.AppendMenu(MF_GRAYED, ID_MENUITEM_SVR_PAUSE, _T("暂停MyWeb"));
            ContextMenu.AppendMenu(MF_STRING, ID_MENUITEM_SVR_STOP, _T("停止MyWeb"));
            break;
        case SERVER_RUNNING:
            ContextMenu.AppendMenu(MF_GRAYED, ID_MENUITEM_SVR_START, _T("启动MyWeb"));
            ContextMenu.AppendMenu(MF_STRING, ID_MENUITEM_SVR_PAUSE, _T("暂停MyWeb"));
            ContextMenu.AppendMenu(MF_STRING, ID_MENUITEM_SVR_STOP, _T("停止MyWeb"));
            break;
        }

        ContextMenu.AppendMenu(MF_SEPARATOR);
        ContextMenu.AppendMenu(MF_STRING, ID_MENUITEM_SVR_EXIT, _T("退出MyWeb"));
```

```
351             SetForegroundWindow();

352             ContextMenu.TrackPopupMenu(TPM_RIGHTALIGN | TPM_RIGHTBUTTON,
                    CursorPos.x, CursorPos.y, this, NULL);
353         }
354 }
////////////////////////////////////////////////////////////////////////////
```

第 81～83 行添加用户自定义消息 WM_USER_SHOWWND、WM_USER_CLIENT 和 WM_USER_NOTIFYICON 的消息处理函数。

第 111～114 行进行额外的对话框初始工作，首先调用 UpdateMenuState 成员函数正确设置主菜单各菜单项的启用/禁用关系（由服务器当前状态决定）；然后创建 CMyNotifyIcon 类的实例 m_pNotifyIcon，传入的参数是当前对话框的句柄 m_hWnd，以及用户自定义消息 WM_USER_NOTIFYICON 和常量 TRAY_ICON_ID；最后调用 m_pNotifyIcon 成员函数添加任务栏图标。

第 126～127 行处理主对话框的最小化消息，选择将其隐藏。

第 161～166 行 OnHlpAbout，protected 成员函数，消息处理函数，显示 About 对话框。

第 167～179 行 OnOptSetup，protected 成员函数，消息处理函数，显示服务器配置对话框（见 22.2.2 节）。如果对话框关闭时返回 IDOK（用户单击了“确定”按钮）并且服务器当前处于 SERVER_RUNNING 运行状态，那么询问用户是否需要重启服务器，并按用户回答进行相应操作。

第 180～194 行 OnOptDefault，protected 成员函数，消息处理函数，复位服务器配置。

第 195～213 行 OnSvrStart，protected 成员函数，消息处理函数。该函数完成了启动服务器（调用全局变量 CHttpServer 类的实例 g_pHttpServer 的相应函数，198～201 行）、刷新主菜单（202 行）、添加服务器信息（203～210 行）以及切换任务栏图标等工作（211 行）。

第 214～226 行 OnSvrPause，protected 成员函数，消息处理函数。该函数完成了暂停服务器（217～220 行）、刷新主菜单（210 行）、添加服务器信息（222 行）以及切换任务栏图标等工作（223 行）。

第 227～238 行 OnSvrStop，protected 成员函数，消息处理函数。该函数完成了停止服务器（230～233 行）、刷新主菜单（234 行）、添加服务器信息（235 行）以及切换任务栏图标等工作（236 行）。

第 239～250 行 OnSvrExit，protected 成员函数，消息处理函数。函数首先判断服务器当前状态，如果处于运行或者暂停状态，则让用户确认是否停止服务。在得到确认后，函数调用 OnSvrStop 函数停止服务器；否则直接返回。在停止服务器后，函数删除任务栏图标，释放 m_pNotifyIcon 所占资源，最后调用基类 CDialog 的 OnCancel 函数进行进一步的资源释放工作。

第 256～278 行 UpdateMenuState，protected 成员函数，根据服务器的当前状态设定主菜单中菜单项的启用/禁用状态。

第 279～289 行 AppendSvrInfo，public 成员函数，将用户信息 strNewInfo 和当前时间组合起来添加到成员变量 m_strServerInfo，然后调用 UpdateData 函数将其显示。

第 290～298 行 OnClientMsg，protected 成员函数，消息处理函数。函数首先接收



CMyHttpServer 模块传来的服务器消息，其中参数 wParam 是连接的客户端的 IP 地址，lParam 是消息的子类型。然后根据该消息及消息参数组成服务器信息，并调用 AppendSvrInfo 成员函数将其显示。

第 303～317 行 OnNotifyMsg，protected 成员函数，消息处理函数，解析并响应用户自定义的任务栏图标消息。接收的两个消息参数 wParam 和 lParam 分别是任务栏图标的 ID 和鼠标事件值。如果是左键双击任务栏图标，那么显示服务器的主对话框（306～308 行）；如果是右键单击任务栏图标，则调用 PopMenu 成员函数显示弹出菜单（309～311 行）。

第 318～354 行 PopMenu，protected 成员函数，显示任务栏图标的弹出式菜单。

22.2.6　其他

此外，MyWeb 项目源码中还包含了类 CAboutDlg 和 CMyWebServerApp，由于这两个类完全由 VC 系统自动生成，这里就不再赘述。

22.3　总　　结

本章给出了一个完整的 WEB 服务器 MyWeb 的源码，结合这些代码向读者展示了如何进行 Winsock 编程、如何进行线程同步，以及如何应用完成端口模型进行复杂的 TCP 服务器设计和实现。总的来说，MyWeb 服务器可以作为大型软件的 WEB 发布模块使用，同时，由于其采用了高性能的完成端口框架，因此也可以扩展为独立的 Win32 平台下的 WEB 服务器。

当然，MyWeb 项目主要用作演示，在编程过程中更多考虑的是程序的简洁性和可读性，因此要成为一个独立的高性能服务器，要做的改进还很多，例如：采用启发式算法替代固定的服务线程数，维持一个打开文件的句柄链表以替代当前的为每一个资源请求都进行文件打开/关闭 I/O 操作的处理方式等，这些都是读者在对 MyWeb 进行改进时需要考虑的。

附录　RFC

本附录对与本书中内容相关的部分 RFC 进行分类，并简要说明其内容，供读者进一步研究相关问题作参考。

1．一般内容

RFC980　　协议文档顺序信息
RFC1009　　互联网网关需求
RFC1011　　官方网际协议
RFC1122　　互联网主机需求-通信层次
RFC1123　　互联网主机需求-应用和支持
RFC1127　　主机需求 RFC 的前途
RFC1173　　主机和网络管理的责任：互联网管理摘要
RFC1175　　用户参考从何处开始，网络信息参考书
RFC1180　　TCP/IP 指南
RFC1206　　问题及回答 FYI：互联网新手问题解答
RFC1207　　问题及回答 FYI：有经验的互联网用户问题解答
RFC1208　　网络术语表
RFC1251　　互联网历史：IAB, IESG and IRSG 成员传记
RFC1340　　分配端口
RFC1360　　IAB 官方协议标准

2．IP 层

RFC791　　网际协议
RFC792　　网际控制消息协议
RFC814　　名字、地址、端口和路由
RFC815　　IP 数据报从组标准
RFC826　　地址解析协议
RFC886　　消息头插入的建议标准
RFC903　　反向地址解析协议
RFC919　　互联网数据报广播
RFC922　　在子网上广播互联网数据报
RFC932　　子网编址计划
RFC950　　子网划分过程互联网标准
RFC1027　　使用 ARP 实现透明子网网关
RFC1088　　在 NetBIOS 网络上传输 I P 数据报的标准
RFC1112　　IP 多点播送的主机扩充

RFC1219　子网号分配

3．路由协议

RFC827　外部网关协议（EGP）
RFC904　外部网关协议正规标准
RFC1058　路由信息协议
RFC1074　基于 SPF 的 NSFNET 骨干网内部网关协议
RFC1163　边界网关协议
RFC1164　边界网关协议在互联网中的应用
RFC1195　使用 OSI IS-IS 在 TCP/IP 和双环境中进行路由选择
RFC1222　NSFNET 路由体系结构扩充
RFC1245　OSPF 协议分析
RFC1246　OSPF 协议使用经验
RFC1247　OSPF 版本 2
RFC1267　边界网关协议 3

4．传输层

RFC768　用户数据报协议
RFC793　传输控制协议
RFC813　TCP 窗口和确认策略
RFC879　TCP 最大段长度及相关主题
RFC896　TCP/IP 网络拥塞控制
RFC1072　长延迟路径 TCP 扩充

5．域名系统

RFC799　互联网域名
RFC920　域需求
RFC974　邮件路由和域名系统
RFC1032　域名管理员指南
RFC1033　域名管理员操作指南
RFC1034　域名—概念和工具
RFC1035　域名—实现和规范
RFC1101　网络名称 DNS 编码及其他编码

6．邮件

RFC821　简单邮件传输协议
RFC974　邮件路由和域名系统
RFC1056　PCMAIL 个人电脑上的分布式邮件系统
RFC1341　MIME（多用途互联网邮件扩充）指定和描述互联网消息体的格式的机制

7．文件传输和文件访问

RFC775　面向目录的 FTP 命令
RFC783　TFTP 协议
RFC949　FTP 惟一的名字存储命令

RFC959 文件传输协议
RFC1094 NFS：网络文件系统协议规范

8．终端访问

RFC854 Telnet 协议规范
RFC855 Telnet 属性规范
RFC856 Telnet 二进制传输
RFC857 Telnet 应答属性
RFC858 Telnet 压缩传输属性
RFC859 Telnet 状态属性
RFC885 Telnet 纪录结束属性
RFC933 输出标记 Telnet 属性

9．网络管理

RFC1155 基于 TCP/IP 互联网络管理信息的结构和标识
RFC1156 基于 TCP/IP 互联网络管理的管理信息库
RFC1157 简单网络管理协议（SNMP）
RFC1212 简明 M I B 定义
RFC1213 基于 TCP/IP 的互联网络管理信息基：MIB-II
RFC1214 OSI 互联网管理：管理信息基
RFC1215 使用 SNMP 定义陷阱的协定

10．Ipv6

RFC1287 未来的 Internet 体系结构
RFC1375 对新的 IP 地址类别的建议
RFC1454 下一版本 IP 提案的比较
RFC1667 IPng 建模和仿真需求
RFC1668 IPng 统一的选路需求
RFC1669 IPng 标准的市场生命力
RFC1670 对 IPng 工程考虑的输入
RFC1671 向 IPng 过渡和其他考虑的白皮书
RFC1675 对 IPng 安全性的关注
RFC1683 IPng 中的多协议互操作性
RFC1726 选择下一代 IP（Ipng）的技术准则
RFC1744 Internet 地址空间管理的观察报告
RFC1752 对 IP 下一代协议的建议
RFC1809 IPv6 中流标记字段的用法
RFC1826 IP 身份验证头
RFC1881 IPv6 地址分配管理
RFC1883 IPv6 技术规范
RFC1884 IPv6 寻址体系结构
RFC1885 用于 IPv6 的 Internet 控制报文协议（ICMPv6）的技术规范

RFC1924　IPv6 地址的一种紧凑的表示方法
RFC2185　向 IPv6 过渡的选路问题
RFC2373　IPv6 寻址体系结构
RFC2374　IPv6 可集聚全球单播地址格式
RFC2375　IPv6 组播地址指派

参考文献

1．Douglas E.Comer 著．林瑶，蒋慧，杜蔚轩等译．用 TCP/IP 进行网际互联（第一卷）：原理、协议与结构（第四版）

2．W.Richard Stevens 著．范建华译．TCP/IP 协议详解：卷一

3．Gary R.Wright W.Richard Stevens 著．陆雪莹译．TCP/IP 协议详解：卷二

4．W.Richard Stevens 著．胡谷雨，吴礼发译．TCP/IP 协议详解：卷三

5．W. Richard Stevens 著．施振川，周利民，孙宏晖等译．Unix 网络编程（第一卷）——套接口 API 和 X/Open 传输接口 API

6．David Bennett 等著．徐军译．Visual C++ 5 开发人员指南

7．MSDN．Microsoft Visual C++ -- Language and Libraries for Visual C++

8．MSDN．Microsoft Platform SDK -- Windows Sockets 2

9．Multithreaded server class with example of HTTP server, Souren Abeghyan, from www.codeproject.com

10．Anthony Jones 著．京京工作室译．WINDOWS 网络编程技术．北京：机械工业出版社

11．Windows Sockets 2.0: Write Scalable Winsock Apps Using Completion Ports, Anthony Jones and Amol Deshpande, MSDN Magazine, Oct 2000

12．Inside I/O Completion Ports, Mark Russinovich, June 30, 1998, from www.sysinternals.com

13．Programming Server-Side Application for Microsoft Windows 2000, Jeffrey Richter and Jason D. Clark, Microsoft Press

教师建议反馈表

1．姓名________ 2．性别________ 3．年龄________ 4．电话________________

5．学校院系________________________________ 6．职务/职称 ________

7．通信地址 ________________________________ 邮编 ________

8．电子信箱 ____________________________ 学校网站 ____________

9．您的文化程度： □大专 □本科 □硕士 □博士

10．您所教学的专业： □计算机类 □数学类 □电子信息 □信息管理类

11．您将本书用作： □本科教学 □自己参考 □学生参考 □其他

12．您的学生层次： □普通本科 □成人教育 □网络教育 □研究生 □社会培训

13．您认为是否需要： □习题解答 □实验指导 □幻灯片 □教师参考书或讲义

14．您从何处了解本书：□教材目录 □他人推荐 □书店 □清华网站 □其他

15．您购买本书在： □新华书店 □校园书店 □科技书店 □教材推广 □邮购

16．您认为本书对应的课程教学有何特点和应注意的地方？

____

____

____

____

17．您最近是否有写作计划，是教材还是一般科技书，针对哪些读者群？

____

____

____

18．您对本书的建议和意见：

____

____

19．您今后需要哪些课程的教材？

____

____

20．您以及您所在学校选用教材有何原则？

____

____

____

表格填好后请寄：

有关计算机/电子/通信类等书籍投稿意向请按照如下方式联系：

地址：北京清华大学校内金地公司 邮编：100084

电话：010-62788951/62791976 转 222 传真：010-62788903

信箱：yuningma@263.net

有关本书的建议和意见或邮购本书请按照以下方式联系：

地址：北京清华大学校内金地公司《TCP/IP 协议及网络编程技术》编辑部

邮编：100084 电话：010-62770384

公司网址：www.thjd.com.cn 传真：010-62788903